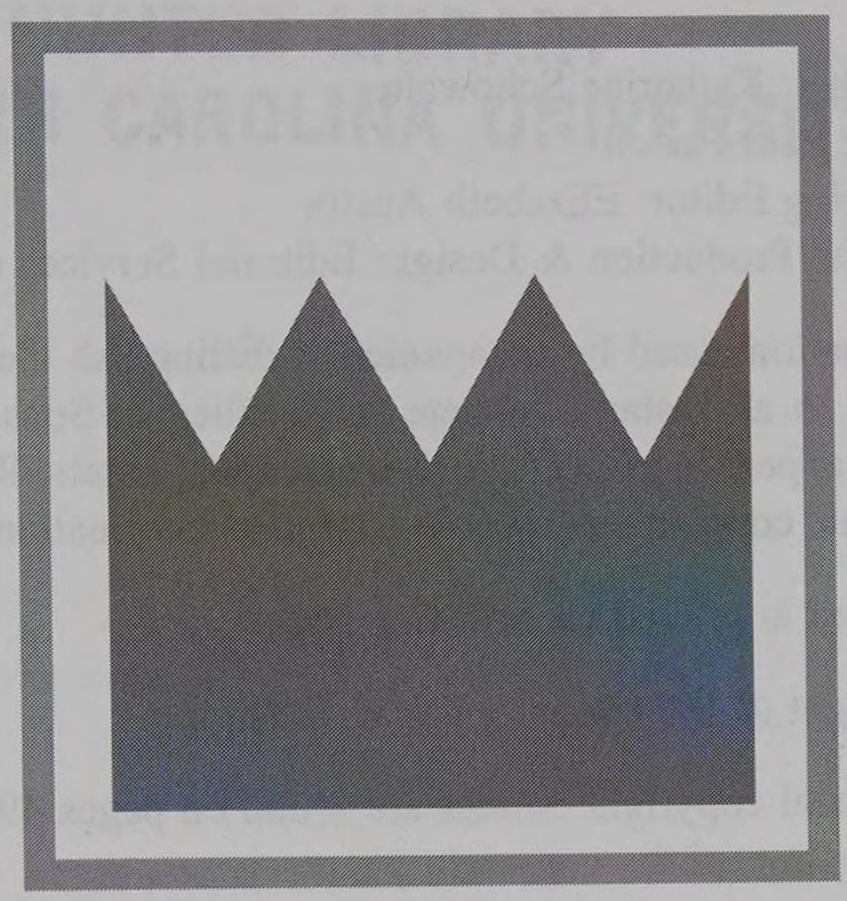
# PROGRAMMING FOR GRAPHICS FILES IN C AND C++

Memory Management • Format Conversions • Color Quantization • File Compression

JOHN LEVINE

DISCARDED BY HUNTER LIBRARY WCU



# Programming for Graphics Files

## In C and C++

John Levine

John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • BRISBANE • TORONTO • SINGAPORE

Publisher: Katherine Schowalter
Editor: Paul Farrell
Managing Editor: Elizabeth Austin
Editorial Production & Design: Editorial Services of New England, Inc.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc. is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This text is printed on acid-free paper.

Copyright ©1994 John Wiley & Sons, Inc.

Additional copyright notices are found on pages 493 and 494 and these pages constitute a continuation of the copyright page.

All rights reserved. Published simultaneously in Canada.

In recognition of the importance of preserving what has been written, it is a policy of John Wiley & Sons, Inc. to have books of enduring value published in the United States printed on acid-free paper, and we exert our best efforts to that end.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought. ADAPTED FROM A DECLARATION OF PRINCIPLES JOINTLY ADOPTED BY A COMMITTEE OF THE AMERICAN BAR ASSOCIATION AND A COMMITTEE OF PUBLISHERS.

Reproduction or translation of any part of this work beyond that permitted by Section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

*Library of Congress Cataloging-in-Publication Data*

Levine, John R.
Programming for graphics files in C and C++ / John Levine.
p. cm.
Includes index.
ISBN 0-471-59854-2 (pbk.). — ISBN 0-471-59856-9 (book/disk), — ISBN 0-471-59857-7 (disk)
1. Computer graphics. 2. C (Computer program language) 3. C++ (Computer program language) I. Title.
T385.L47 1993
006.6'6–dc20 93-31897
CIP

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

HUNTER LIBRARY
WESTERN CAROLINA UNIVERSITY

# Contents

Preface xi

*Who This Book is For* xi
*How to Use This Book* xi
*Origin of the Source Code in the Book* xii
*Your Opinion is Important* xii

PART ONE
Graphics Files in General 1

CHAPTER 1
What Are Graphics Files? 3

1.1 File Encodings 3
1.2 Image Geometry 5
*Screen Coordinates* 5
*Graph Coordinates* 6
*Other Coordinate Systems* 6

CHAPTER 2
Types of Graphics Files 7

2.1 Bitmap Files 7
*Pixel Ordering* 7
*Scan Lines* 7
*Bit Planes* 9
*Interleaved Rows* 9
*Lossy Compression* 10
2.2 Vector Files 10

2.3 Metafiles 10
2.4 Printer Data Files 11
*Extended Text Formats* 11
*Page Description Languages* 11
2.5 Converting Among File Types 12
*Bitmap to Bitmap* 12
*Vector to Vector* 12
*Vector to Bitmap* 12
*Bitmap to Vector* 13

CHAPTER 3
Colors 14
3.1 Monochrome versus Color Images 14
3.2 Color Models 14
*Luminance/Chrominance Models* 15
3.3 Color Maps 15

CHAPTER 4
Compression Techniques 17
4.1 Run-Length Encoding 17
4.2 Huffman Coding 18
4.3 Dictionary Systems 18
4.4 More Complex Schemes 19
*Arithmetic Coding* 19
*Lossy Compression* 19

CHAPTER 5
Memory Management for File Processing 21
5.1 Handling Small and Medium Files 21
5.2 Handling Large Files in a Virtual Environment 23
5.3 Handling Large Files in an MS-DOS Environment 23
*Virtual Segmented Pixel Arrays in C++* 24
5.4 Handling Partial Files 30
5.5 Techniques for Vector and Metafiles 30

PART TWO
A Framework for Bitmap Files 33

CHAPTER 6
A Framework for Bitmap Files 35
6.1 The PBM Approach 35

6.2 PBM Formats 36
*Single-Bit PBM Formats* 36
*Grayscale PGM Formats* 36
*Color PPM Formats* 37
*General Formats* 37
6.3 Support Libraries 38
*Header Files* 38

CHAPTER 7
Libraries for Monochrome Images 43

7.1 PBM Routines 43
7.2 Memory Allocation Routines 45
7.3 Program Startup Routines 48
7.4 Error Reporting 51
7.5 Miscellaneous Support 53
7.6 File Handling 54
7.7 Integer Read/Write Routines 55
7.8 Reading PBM Format Files 59
7.9 Low-level Header Reading Routines 62
7.10 Writing PBM Format Files 64

CHAPTER 8
Libraries for Grayscale Images 68

8.1 PGM Headers 68
8.2 PGM Initialization 70
8.3 Reading PGM Files 71
8.4 Writing PGM Files 74

CHAPTER 9
Libraries for Full-Color Images 79

9.1 PPM Initialization 82
9.2 Reading PPM Files 83
9.3 Writing PPM Files 88
9.4 Color Maps 92
*Color Map Data Structures* 92
*Color Map Building Routines* 94

CHAPTER 10
Libraries for General Pixel Images 102

10.1 PNM Headers 102
10.2 PNM Initialization 104
10.3 Reading PNM Files 104
10.4 Writing PNM Files 107

PART **THREE**
**Run Length Compressed Formats** **113**

CHAPTER **11**
**MacPaint** **115**

11.1 Overview 115
11.2 Reading MacPaint Files 116
11.3 Writing MacPaint Files 117

CHAPTER **12**
**PCX** **122**

12.1 Reading PCX Files 123
*Uncompressing the Stored Image Data* 128
*Converting Data Planes to Pixels* 129
*Unpacking Packed Pixels* 130
12.2 Writing PCX Files 132
*Encoding a PCX File* 135
*Image Encoding Helper Routines* 139
*Compressing Plane Data* 141
12.3 Possible Extensions 143

CHAPTER **13**
**GEM IMG** **143**

13.1 IMG Files 143
13.2 Reading IMG Files 143
13.3 Writing IMG Files 148
*Bit Packing and Writing* 151
13.4 Possible Extensions 153

CHAPTER **14**
**IFF/ILBM Bitmap Images** **154**

14.1 ILBM Header 155
14.2 Reading ILBM Files 156
*Reader Auxiliary Routines* 165
14.3 Writing ILBM Files 166
*ILBM Writing Auxiliary Routines* 173

PART **FOUR**
## Uncompressed Formats 177

CHAPTER **15**
### Windows Bitmap 179

15.1 Common BMP Header 179
15.2 Reading BMP Files 180
*Decoding BMP Files* 181
15.3 Writing BMP Files 185
*Encoding BMP Files* 188

CHAPTER **16**
### Targa 192

16.1 Targa Include File 192
16.2 Reading Targa Files 193
*Auxiliary Routines for Reading Targa Files* 199
16.3 Writing Targa Files 203
*Auxiliary Routines for Writing Targa Files* 210

PART **FIVE**
## Dictionary Compressed Formats 215

CHAPTER **17**
### GIF 217

17.1 Writing GIF Files 218
*GIF Encoding* 222
*LZW Compression* 228
17.2 Reading GIF Files 239
*Decoding the Compressed Image* 246

CHAPTER **18**
### TIFF 255

18.1 TIFF Subsets and Classes 255
18.2 The TIFF Library 256
*Using the TIFF Library* 256
*Library Headers* 257
*Handling IFDs* 276
18.3 Handling TIFF IFDs 298
*Reading TIFF IFDs* 298
*Writing TIFF IFDs* 320
18.4 Opening and Closing TIFF Files 335

*Closing TIFF Files* 343
18.5 Compression and Encoding Techniques 345
*No Compression* 350
*PackBits Compression* 354
*Other Compression Methods* 360
18.6 Reading TIFF Data 361
18.7 Writing TIFF Data 368
*Other Odds and Ends* 377
18.8 TIFF Reading and Applications 383

CHAPTER 19
JPEG 385

19.1 What Is JPEG? 385
*JPEG Formats* 386
19.2 Steps in JPEG Encoding 386
*Color Coding* 387
*DCT Transformation* 387
*Quantization* 388
*Huffman or Arithmetic Coding* 388
19.3 Using the JPEG Library 388
*Standalone Programs* 389
*Library Code* 389
*JPEG Compression* 389
*JPEG Decompression* 395

PART SIX
Vector Formats 405

CHAPTER 20
HP-GL 407

20.1 Reading HP-GL Files 407
20.2 Low-Level HP-GL Reader 408
20.3 High-Level HP-GL Reader 414

CHAPTER 21
MS Windows Metafiles 418

21.1 Reading and Writing Under MS Windows in C 418
21.2 Under MS Windows in C++ 419
21.3 Reading Metafiles in C 420

PART **SEVEN**
**Printer Formats** 429

CHAPTER **22**
HP Printer Control Language 431

CHAPTER **23**
PostScript 436

23.1 Writing PostScript 436
23.2 Reading PostScript 451
*Reading EPS Preview Data* 451

PART **EIGHT**
**Format Conversion** 455

CHAPTER **24**
Bitmap to Bitmap: The PBM Approach 457

24.1 Using PBM Programs Together 457
24.2 Bitmap Transformations 457
*Color to Gray Conversion* 458
*Color Quantization* 458
*Dithering* 459

CHAPTER **25**
Vector to Bitmap: Rasterization 462

25.1 Rasterization Library 462
25.2 HPGL to PPM 470
25.3 Extending the Rasterizer 474

CHAPTER **26**
Guessing a File Format 475

References 477
About the Software 478
Index 481

# Preface

Graphics have become ubiquitous on computers, ranging from the smallest PC graphing package to the largest supercomputer visualization system. Any graphic image that is worth displaying is worth saving, and to save an image, an application needs a graphics file. As graphics systems have evolved, dozens of file formats have evolved as well. It has become an unfortunate fact of life that any application that loads or stores any sort of graphics file has to be able to handle a variety of formats so that users will be able to handle the files they already have.

## WHO THIS BOOK IS FOR

The primary audience for this book is programmers who need to import or export graphics files to their applications. It also shows how to print bitmapped formats, a task that has a lot in common with writing files.

For the simpler formats (all of the bitmap formats except JPEG), the book includes complete reading and writing files in the desired formats, which programmers should be able to adapt into their applications. Reading and writing JPEG is so complicated that there isn't room to show all of the code in the book, so the JPEG library code is on the companion disk, and the text shows how an application interfaces to it. If you purchased this book without the companion disk, see the ordering form at the back.

Programming managers should also find the book useful to understand how much work is involved in adding support for various formats to an existing or proposed application.

## HOW TO USE THIS BOOK

Please at least skim the first five chapters in the book before diving into the code. They provide essential background information necessary to understand the code.

All of the bitmap code except TIFF and JPEG depend on the PBM libraries developed in Chapters 6 through 10. Again, please look at them before diving into particular bitmap formats.

The rest of the chapters stand on their own (except for a few exceptions noted in the text). In particular, none of the code for the bitmap formats depends on each other, so the reader can look at just the formats of particular interest.

## ORIGIN OF THE SOURCE CODE IN THE BOOK

Most of the code in this book is extracted from publicly available libraries that have been refined and tested over several years.

The bitmap libraries and the code for MacPaint, PCX, GEM IMG, IFF/ILBM, Targa, GIF, and the TIFF application are adapted from the PBMPLUS library written by Jef Poskanzer and others. The version in the book has been enhanced so that it runs on MS-DOS as well as on Unix and VMS and has had some bugs and inefficiencies fixed. (The companion disk includes DOS-compatible versions of the entire PBMPLUS suite, not just the programs displayed in the text.) The TIFF library code is by Sam Leffler of Silicon Graphics, with minor modifications for MS-DOS compatibility. The JPEG code is from the Independent JPEG Group, headed by Tom Lane.

The BMP reading and writing code and all of the C++ code is new for this book.

## YOUR OPINION IS IMPORTANT

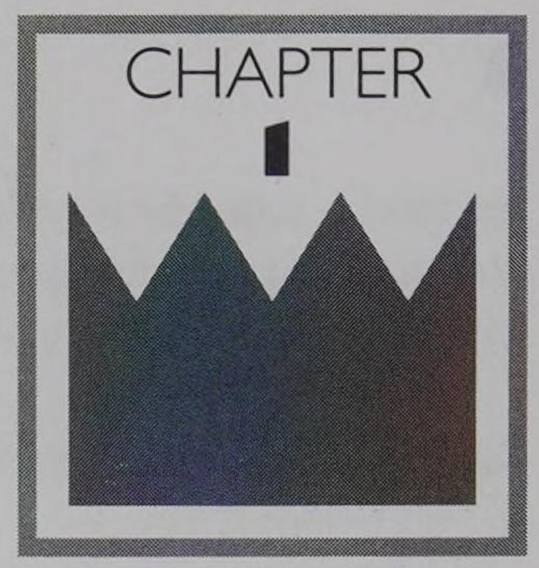
All authors love to hear from their readers. If you liked this book, or particularly if you didn't, please drop me a line at

John R. Levine
I.E.C.C.
Post Office Box 380349
Cambridge, MA 02238-0349

Readers with electronic mail access can send mail to the Internet address graphics@iecc.com.

John Levine
Cambridge, Mass.
May 1993

# Graphics Files in General




# What Are Graphics Files?

A graphics file is no more than a file that contains a picture. This covers a lot of possibilities, ranging from a line drawing like Figure 1.1 to a scanned photograph like Figure 1.2 (both on the next page). Any program that displays stored images needs to be able to read in a graphics file, and any program that manipulates images also needs to be able to store images away for later use.

Although there are only a few fundamentally different ways to encode an image, something we discuss at length in Chapter 2, there are dozens of incompatible file formats, each with its own peculiarities. We will start with some general structure to handle each kind of file format, then look in detail at a variety of specific formats to see what is involved in reading or writing each of them.

## 1.1 FILE ENCODINGS

Data in a graphics file can be encoded in many different ways. At one extreme, the information can be readable ASCII text, which is easy for humans to read and, to some extent, to edit, and easy for programs to read and write. But it is bulky and slow to read and write from programs. At the other extreme are compressed binary formats, which are very compact but incomprehensible to humans and require complex reading and writing routines.

Formats also vary greatly in how much flexibility they offer in terms of image size, shape, colors, and other attributes. At one end is MacPaint, which allows exactly one size of picture in exactly two colors stored in exactly one way. At the other is TIFF, the Tagged Image File Format, with so many different options and features that no existing TIFF implementation can read all of them.

As we will see in Chapter 2, file formats fall into several general categories. The most common are *bitmap* formats, which represent a picture as a rectangular array of dots, reminiscent of a television image. Somewhat less common

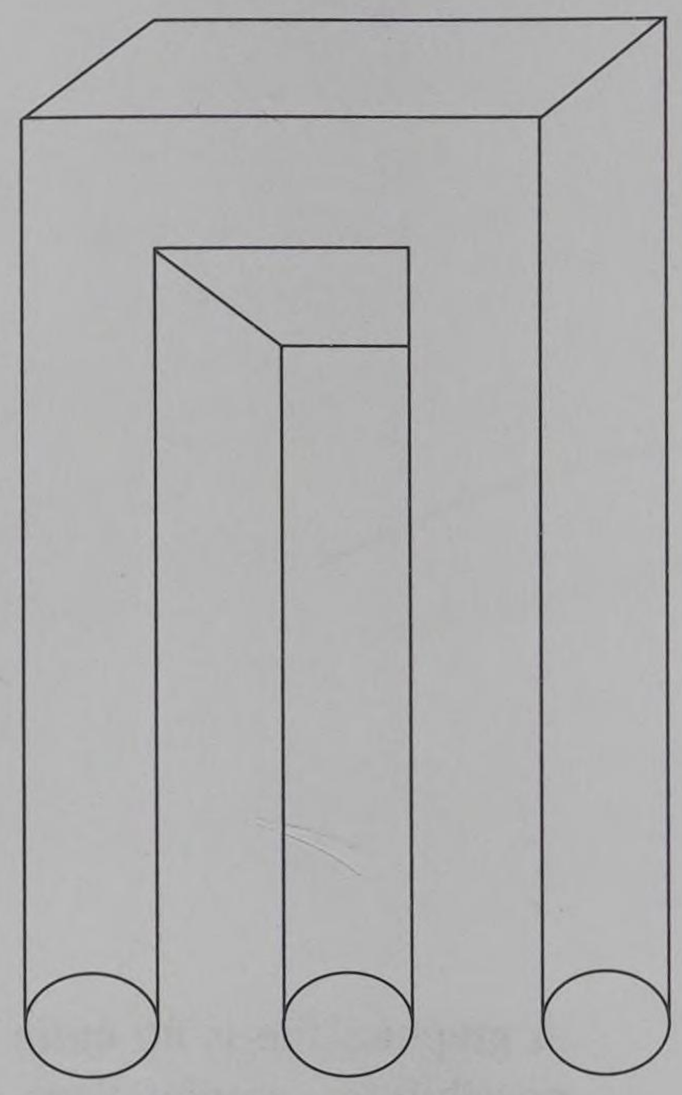


**Figure 1.1**
A line drawing.

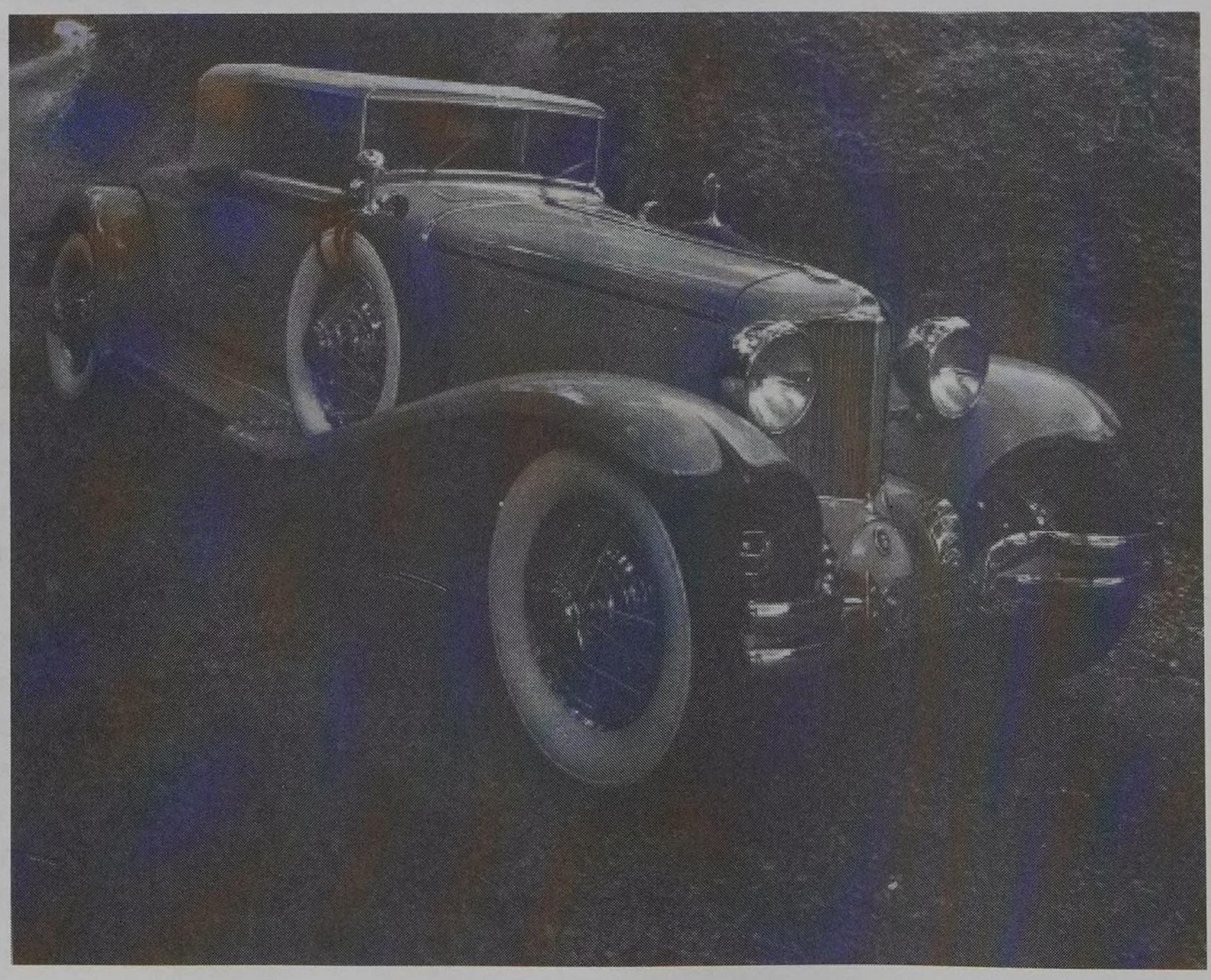


**Figure 1.2**
A scanned photograph.

are *vector* formats, which represent a picture as a series of lines and arcs. Complex graphics systems provide *metafiles*, which are a list of commands to the graphics system. And for today's sophisticated laser printers, the data stream necessary to print an image on the printer has become a format in its own right.

Within each format category, there is a set of techniques applicable to all formats in the category.

## 1.2 IMAGE GEOMETRY

Every computer image has an internal geometry used to position elements in the picture. Fortunately, the number of different geometries used is small. All use a "graph paper" model, with positions in the picture specified by $x$ and $y$ coordinates relative to an origin. The $x$ coordinate is the horizontal distance from the origin, and the $y$ coordinate is the vertical distance. The two most common versions of this model are what one might call *screen coordinates* and *graph coordinates*.

### Screen Coordinates

Screen coordinates, as shown in Figure 1.3, are most commonly used in images intended for display on a video screen. In conventional notation, the coordinates are expressed in parentheses as follows: $(x,y)$. The (0,0) point is at the upper left corner of the image, $x$ coordinates increase to the right of the origin, and $y$ coordinates increase below the origin. For example, (1,0) is the point immediately to the right of the origin, and (0,1) is the point immediately below the origin.

Usually, one unit of $x$ distance is the same as one unit of $y$ distance. In some cases, though, the two scales are different. Most often this is because the image was saved from a screen that distorts the image, such as an IBM CGA or VGA or an older Amiga screen.

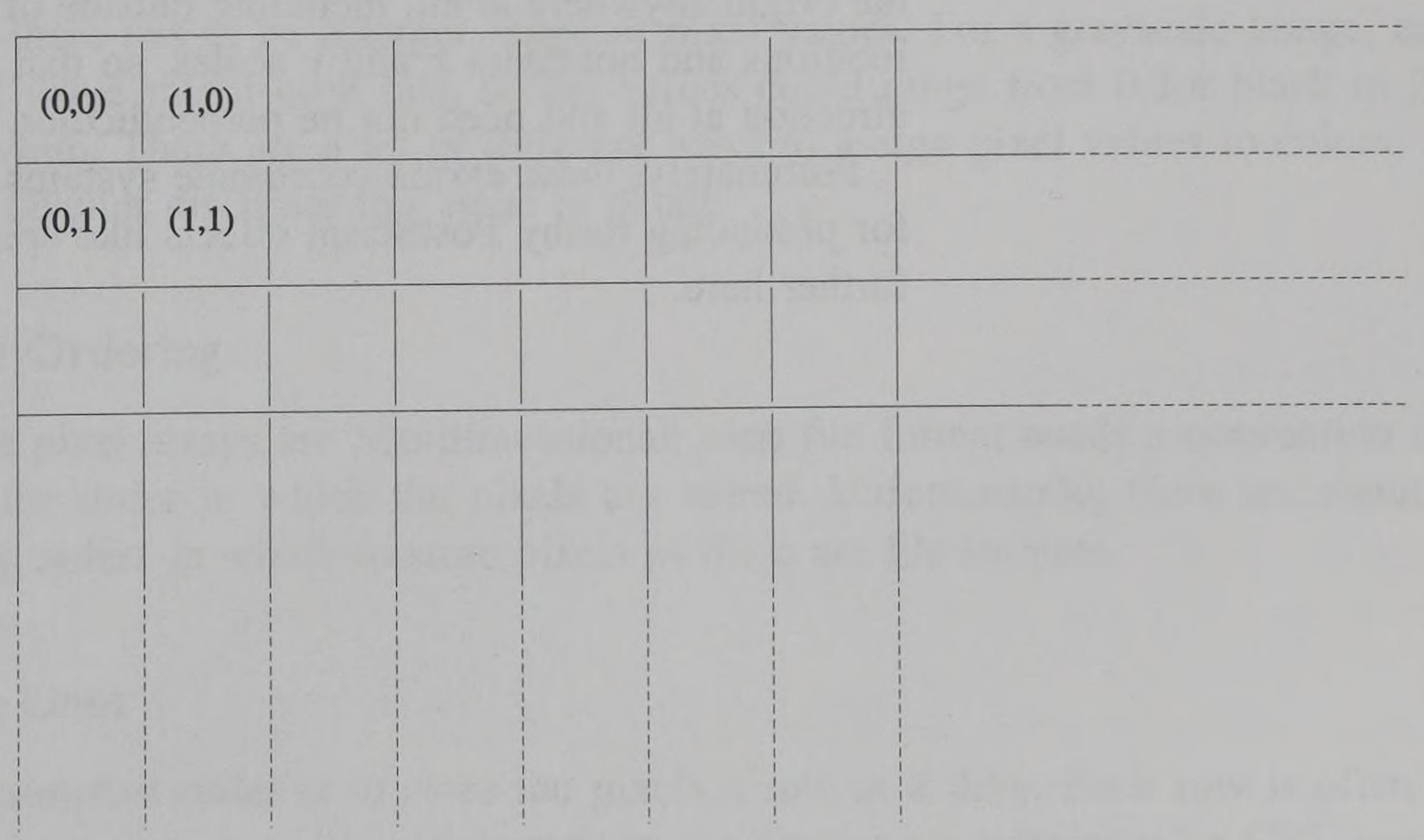



**Figure 1.3**
Screen coordinates.

## Graph Coordinates

Graph coordinates, as shown in Figure 1.4, are most often used in images intended to be printed on a piece of paper. The origin is at the lower left of the image. *x* coordinates increase to the right, and *y* coordinates increase above the origin. (This coordinate system is the one that most of us learned in high school geometry.)

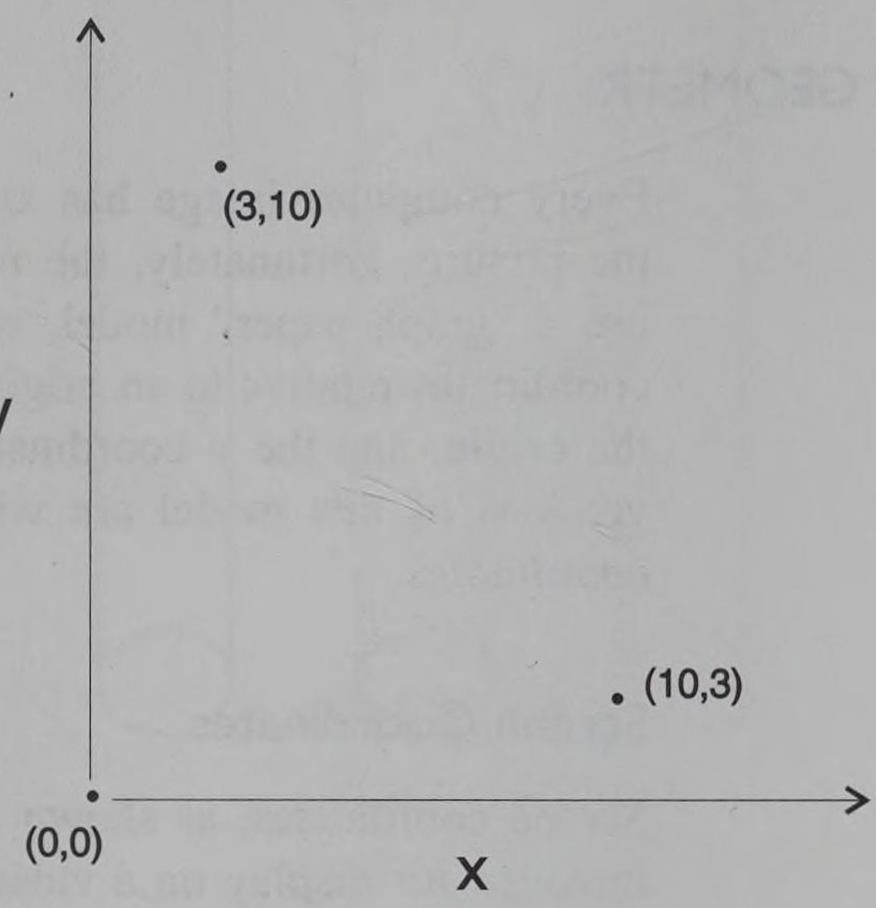
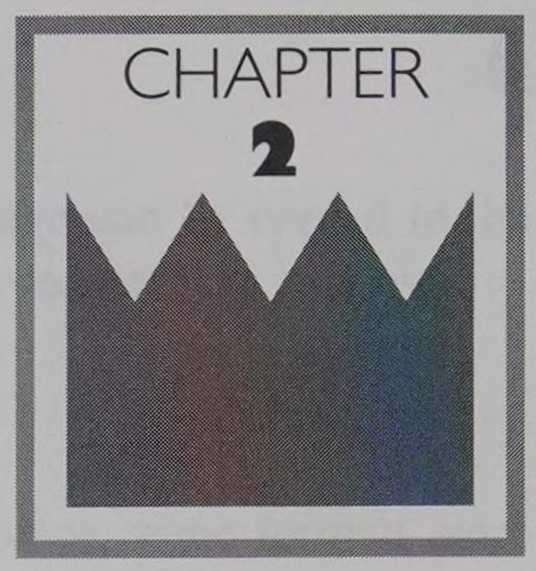



**Figure 1.4**
Graph coordinates.

## Other Coordinate Systems

Occasionally you may find an image with the origin at a different point, such as the upper right or lower left of the image. Some file formats, notably PostScript and HP-GL allow the programmer to set an arbitrary coordinate system with *x* increasing either to the left or right, *y* increasing either to the top or bottom, and the origin anywhere at all, including outside of the image. PostScript even allows rotations and nonequal *x* and *y* scales, so that the *x* and *y* axes can point in any direction at all and need not be perpendicular.

Fortunately, these exotic coordinate systems seem not to be very useful except for producing flashy PostScript effects like spiraling text, so we don't treat them further here.




# Types of Graphics Files

All graphics files fall into a small set of categories. In this chapter we discuss some common characteristics and the types of files before we dive into the details of each individual type.

## 2.1 BITMAP FILES

The most commonly encountered kind of graphics files are *bitmaps* or *raster images*. A bitmap image is a rectangular array of points, known as picture elements or *pixels,* as shown in Figure 2.1. Each point in the image is stored as a number which represents the color of that point, the *pixel value* for that point. In the simplest pictures, each point is either black or white so the pixel value is either 0 or 1, a single bit. More commonly, the picture is in grayscale or color, in which case there has to be a larger range of pixel values. For a grayscale image, each pixel value might be 8 bits, so the values could range from 0 for black to 255 for white. There are a lot of different ways to assign pixel values to colors; the next chapter discusses this issue in detail.

### Pixel Ordering

Since pixel arrays are two-dimensional, each file format needs a convention that sets the order in which the pixels are stored. Unfortunately, there are about as many orders in which to store pixels as there are file formats.

### Scan Lines

The simplest order is to store the pixels a row at a time. Each row is often referred to as a *scan line,* analogous to the lines on a television or CRT screen.

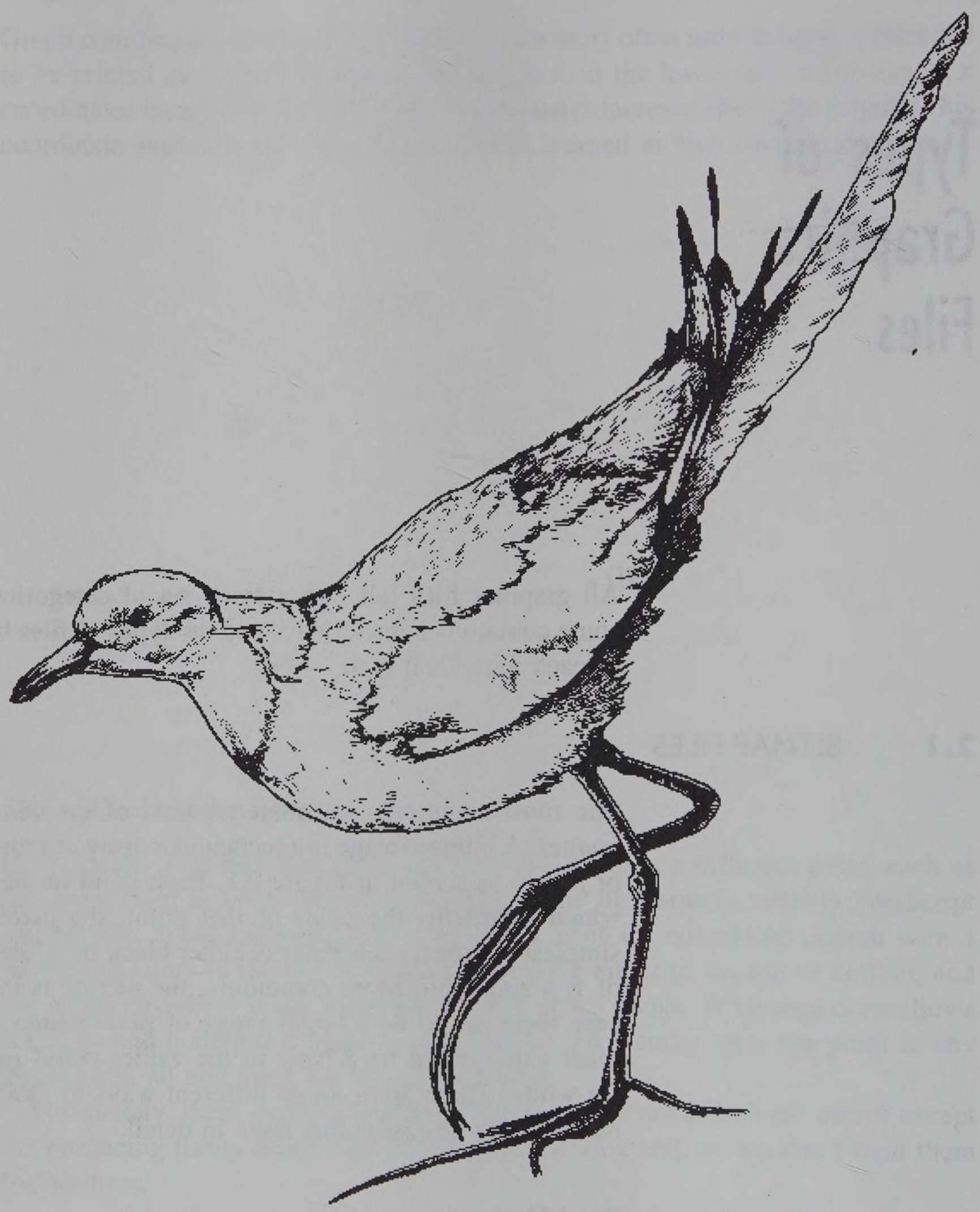


Figure 2.1
A bitmap.

Most often, the pixels in a row are stored from left to right, with the rows from top to bottom. But not always—sometimes the pixels go from right to left, and sometimes the rows go from bottom to top. (These other orders most often result when an image is produced by a scanner.)

When each pixel is smaller than a byte, there is also the question of whether the leftmost pixel in a byte is in the high order or low order bit(s) of the byte. This decision is most often made to agree with the data format used by the video

adapter on the original author's computer, since an image can be copied to the video adapter much more quickly if there is no need to reorder bits in a byte.

### Bit Planes

When the pixels in an image contain more than 1 bit apiece, some formats store the data as *bit planes*. That is, first they store the high order bit of all the pixels, the first bit plane. Then they store the second bit of the pixels, the second bit plane, and so forth, with a bit plane for each bit of the pixels. An image with 6-bit pixels would be stored as 6 separate bit planes. Sometimes an image is stored by plane for each row, rather than by plane for the whole image.

This peculiar-sounding scheme actually has considerable logic behind it. For one thing, it matches the data formats used by a lot of older video adapters. Video adapters stored each bit plane in a separate memory bank. Because of the way that memory chips are built, if you have, say, 6 bit pixels, it is a lot easier in hardware to retrieve simultaneously 1 bit from each of six banks than it is to retrieve 6 bits from one bank, and video adapters have to retrieve all of the bits for each pixel as the pixel is displayed.

Another reason is that it is sometimes useful to manipulate data one plane or a few planes at a time. In some formats, one of the bits in each pixel indicates that the pixel is supposed to be replaced by a background color or otherwise treated specially. If the bits are stored a plane at a time, all of the "special treatment" bits are collected together, which can make it easier to find and operate on the special pixels.

Finally, it can be useful to handle some but not all of the bit planes in a file. If a grayscale picture has 8 bits per pixel, the display device may handle only 6 bits per pixel. The easiest way to convert the image is to use the high 6 bits of each pixel and discard the 2 lower ones. If the image is stored plane by plane, a program can read in the 6 planes of interest without having to read the other 2.

In color images where the pixel values in an image are looked up in a color lookup table, there are other tricks one can perform with bit planes. We discuss these further in Chapter 3.

### Interleaved Rows

Most commonly the rows of pixels in an image are stored in order from the top to the bottom. In some cases, though, the rows are *interleaved*. The simplest interleave method is to store the even numbered rows, then the odd numbered rows, so the rows will be stored in the order 0, 2, 4, . . . , 1, 3, 5, . . . . In some cases, the odd numbered rows would be in the reverse order, so if there were a total of 100 rows, the order would be 0, 2, 4, . . . , 96, 98, 99, 97, 95, . . . , 5, 3, 1.

The original point of interleaving was to match some kinds of video displays. Even-down-odd-up order matches the order of scan lines used on TV sets and on interleaved video displays. Another advantage of interleaved files is that one can construct an approximate version of the image without having to read the whole

file, something that can be particularly useful when a file is being displayed as it is received over a slow network or phone line.

More complex interleave schemes are also possible. GIF format uses a four-way interleave that first stores every eighth row, then three more sets of rows, each of which fills in the rows halfway between the ones already stored.

### Lossy Compression

The JPEG format, although fundamentally a bitmapped format, doesn't store individual pixels. Instead, it stores blocks of data that can be used approximately to reconstruct blocks of pixels. The format is called *lossy* because the reconstructed pixels are not exactly identical to the originals. But the differences will be small, and particularly when the original was a digitized photograph (as opposed to a computer-generated image), the reconstructed image is hard to tell from the original. We discuss JPEG in detail in Chapter 19.

## 2.2 VECTOR FILES

A fundamentally different and much older way to store graphic images is as *vectors*. Rather than storing a value for each point in an image, vector formats store higher-level constructs such as lines, arcs, circles, and text. A file might store a sequence of commands such as these:

```
move to (100,100)
select color blue
line to (100,200)
circle with radius 50
move to (50,50)
select color red
text "This is a label"
```

Vector files are invariably more compact than the equivalent bitmap files, since only the interesting parts of the image need be stored, with everything else implicitly being blank. A vector file is more amenable to higher-level processing than is a bitmap file, since the components of an image are much easier to identify.

The main disadvantage of vector files is that they can only reasonably describe a line drawing, not a photographic image.

## 2.3 METAFILES

A *metafile* is like a vector file, only more so. Along with vectors and circles, metafiles can contain much more sophisticated image creation commands, e.g., define a "brush" as a small bitmap and use that brush to outline or fill an area.

Metafiles are really programs for sophisticated graphics packages. For example, Microsoft Windows metafiles, which we discuss in Chapter 21, are lists of calls to the screen-drawing routines that Windows provides. This means that although it is reasonably easy to create or decode a metafile, drawing the image it describes is impractical without access to the graphics package it depends on.

## 2.4 PRINTER DATA FILES

Modern printers can print very complex pictures, in some cases rivaling typeset or photographic quality. The formats of the data stream that one sends to a printer is itself a file format. There are two general types of printer formats, *extended text formats* and *page description languages*.

### Extended Text Formats

Extended text formats embed picture information inside a conventional text stream. Plain text prints as itself; escape sequences introduce nontext elements. The most widely used extended text format is Hewlett-Packard's Printer Control Language (PCL), Chapter 22, which has become a de facto standard for low- to medium-performance laser printers.

### Page Description Languages

The other approach to controlling a printer is to define an entirely new language to describe what is to be printed on the page. Several such languages have been used in printers in the past, but now PostScript has become the standard description language.

A PostScript printer file is actually a computer program in the PostScript language which draws the desired image. The language is actually a general-purpose computer language somewhat reminiscent of FORTH. One could in principle write entire graphics applications in PostScript, though for practical reasons (most implementations are pretty slow for general calculations), people generally only write code directly related to picture generation.

PostScript programs draw pictures in two general ways. The easier way is simply to draw a pixel map. One of the basic PostScript operators reads in a sequence of pixel values and displays them as a rectangle on the page. This approach is appropriate for including scanned or ray-traced images in documents or any image that is already in bit-mapped form.

Alternatively, one can use PostScript like a vector or metafile format. There are basic PostScript operators that draw lines, circles, curves, boxes, etc., and one can build up a picture from these elements.

One can also combine the two approaches. For example, one could use line, box, and curve operators to define an oddly shaped area, then use that area as a "clip mask" that controls which parts of a rectangular bitmap are actually printed.

PostScript programming is a complex enough topic to merit entire books on its own. Here, we limit the discussion to basic techniques to turn a bitmap file into

a PostScript printer file, and to extract a bitmap from a conventionally formatted PostScript file.

## 2.5 CONVERTING AMONG FILE TYPES

Given that there are so many different file formats, it is inevitable that one will need to convert a file from one type to another. Depending on the types, these conversions can be anywhere from trivial to impossible.

### Bitmap to Bitmap

Converting one type of bitmap file to another is usually easy. Once one handles the details of file coding, all pixels are pretty much the same, making the conversion straightforward.

When the conversion goes from a more expressive format, e.g., full color, to a less expressive one, e.g., black and white, or grayscale, there are well-understood techniques to convert the image while maintaining the best possible image quality. We touch on these techniques in Chapter 24.

### Vector to Vector

The primary issues in converting among vector formats is reconciling the slightly different semantics of differing formats and, to some degree, handling coordinate systems. In the simplest case, commands translate one-to-one, e.g., a "line" command in the original turns into a "line" command in the translated file. Problems arise when the two formats don't have corresponding commands. If the original format has an "ellipse" command, and the target format doesn't, there are a variety of unsatisfactory translation techniques. One might approximate it with a circle, or simulate it with a series of short line segments. Or perhaps the target format allows separate $x$- and $y$-axis scales, in which case one can draw an ellipse by setting different scales and drawing a circle that is scaled into an ellipse.

Vector-to-vector conversion comes up much less than other types, so we don't address it further here.

### Vector to Bitmap

Vector to bitmap conversion involves *rasterization*, finding the set of pixels corresponding to each vector in the original. The basic algorithm for rasterization has been known since 1963, when it was published by Bresenham; circles and arcs can be drawn using modified versions of the same method.

This kind of conversion involves almost exactly the same issues as drawing a vector image on a raster screen or on a laser printer (since both of them really only display arrays of pixels) so that frequently it is possible to adapt display code into bitmap conversion code.

### Bitmap to Vector

Bitmap to vector conversion is far, far more difficult than any of the previous types. Moderately satisfactory *edge-detection* algorithms exist to handle the simplest cases of finding a clearly drawn vector in a bitmap, but the general problem of finding the lines in a scanned image (most typically a faxed document) remains unsolved, and exceeds what we can cover here.

CHAPTER 3

# Colors

All but the simplest images are in color. For vector images, the color scheme is generally pretty simple—there is a small set of available colors, and between vectors one can switch colors. (The most common vector format was designed for pen plotters that had a small set of colored pens available.) But for bitmap and other kinds of images, the color scheme can be quite complex, in order to provide photorealistic color.

In the rest of this chapter, we concentrate on bitmap images, though nearly all of the discussion applies equally to metafiles.

## 3.1 MONOCHROME VERSUS COLOR IMAGES

The simplest distinction is between black-and-white and color images. Black-and-white images are sometimes further divided into monochrome images that use only the two colors black and white, and grayscale, which use more shades of gray.

As we discussed in Chapter 2, most bitmap images actually use more than 1 bit per pixel. For grayscale images, the bits are interpreted as a number, usually with 0 being black and the highest value (255 for 8-bit pixels) being white. Intermediate values represent intermediate shades of gray. Occasionally 0 is white and the highest value is black, again with intermediate values representing intermediate shades. For color images, the pixel bits are interpreted in more complex ways, which we consider next.

## 3.2 COLOR MODELS

There are dozens of different ways to assign numerical values to colors, known as *color models*. The most common model found in bitmap images is the RGB or *additive* model, which is the model used by nearly every video screen controller.

Each pixel consists of three components, numbers representing the amount of red, blue, and green in the pixel. The particular color is created by starting with black and adding in the given amounts of the three component colors. For each component, 0 means none of the color and the maximum value is the full amount of the color. If there are 8 bits per component, (0,0,0) means black, (255,0,0) means bright red, and (255,255,255) means white. Each pixel would be stored as 24 bits.

A related scheme is CMYK or *subtractive*. It uses four components, Cyan, Magenta, Yellow, and blacK, and works by starting with white and then subtracting the given amount of each of the component colors. This model matches the way that colored inks combine on a printed page, so CMYK is used primarily for producing images to be sent to a color printer.

### Luminance/Chrominance Models

Another approach to color coding pays more attention to the way that the human eye sees color. It turns out that the eye is quite sensitive to changes in brightness but much less sensitive to changes in color. This means that small errors in color are nowhere near as noticeable as errors in brightness, so coding schemes can take advantage of this fact to encode pictures in a way that preserves an accurate-looking image while using fewer bits than an RGB scheme.

There are dozens of minor variations on this theme, mostly based on various television signaling schemes. The scheme usually used by JPEG, which is typical, has three components, called Y, Cr, and Cb. The Y component is the *luminance*, essentially the brightness, and the Cr and Cb are the *chrominance*, the redness and blueness. JPEG subsamples the Cr and Cb, meaning that for each $2 \times 2$ block of $y$ values, it stores only one Cr and one Cb value. This hardly affects the appearance of the reconstructed picture, but saves a great deal of space. For each $2 \times 2$ block, instead of storing 12 values (four each of Y, Cr, and Cb) it only stores six (four Y, one Cr, one Cb), which is a 50-percent space saving. JPEG goes on to do some other compression tricks unrelated to color coding. See Chapter 19 for details.

This kind of coding works extremely well on digitized photographs, and fairly well on ray-traced pseudophotographs. It works less well on nonphotographic images with sharp edges, such as those that contain text, because the blurring of the colors is more evident.

In graphics files, luminance/chrominance coding is standard on JPEG files, both standalone JFIF files and JPEG inside TIFF files, and is one of many color options in PostScript.

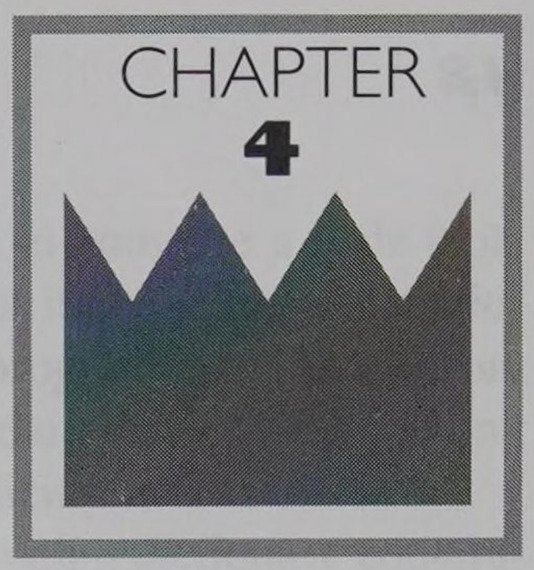
## 3.3 COLOR MAPS

Full-color images can be very large. A $600 \times 800$ image is 480,000 pixels. If each pixel were stored as a 24-bit RGB value, the image would consume 1.4 megabytes. To decrease the amount of space needed, many formats use *color maps* or *palettes*. Rather than storing the actual color for each pixel in the file,

the color map contains a list of all the colors used in the image, and the individual pixel values are stored as entry numbers in the color map. A typical color map has 16 or 256 entries, so each pixel value is only 4 or 8 bits, an enormous savings from 24 bits per pixel. Even though a full-color 24-bit image has far more than 256 colors, if the color map values are well chosen, the actual pixel values can be approximated well enough that the displayed image is of acceptable quality.

Color maps are widely used in screen controllers to minimize the amount of RAM needed in the screen buffer. Programs can create various screen effects by changing the color map. The on-screen color of each pixel that references a particular color map entry changes when the entry changes, so programs can rapidly change screen colors with color map changes. Programs can encode multiple images by treating the bits in pixel entries as a set of bit fields. Suitable settings of the color map can make different images appear or disappear. For example, consider an image with 4-bit pixels. A pair of four-color images can be encoded by using the high 2 bits of each pixel for the first image and the low 2 bits of each pixel for the second image. A suitable color map could make either image visible or allow various overlay effects, for example, just the red pixels in the first image overlaid on the entire second image.

Color-mapped images are often stored a single plane of bits at a time, in order to facilitate this sort of image manipulation.




# Compression Techniques

Since graphics images tend to be so large, a variety of compression schemes are used to make the stored images smaller. Although the details vary from format to format, a few schemes appear in many different formats, so we discuss them here.

## 4.1 RUN-LENGTH ENCODING

The most common compression schemes are variations on *run-length encoding*. These schemes take advantage of the fact that images, particularly computer-generated images, tend to have areas of solid color, meaning that there are often many pixels in a row with the same pixel value. If, for example, the data in a file were abccccddde, an encoded form would be something like 1a1b4c3d1e.

The most common form of run-length encoding is PackBits, originally used in the simple image format stored by MacPaint and now found in about a dozen different formats. PackBits stores repeated byte values with 2 bytes: the first is a count byte, which is the value $257 - r$ where $r$ is the number of repeats. The value of $r$ can range from 2 to 129, so the coded value in the length byte is between 128 and 255. The second byte is the value to repeat. PackBits also has provision for strings of nonrepeated bytes. If the value of the length byte is between 0 and 127, it is considered to be $l - 1$, where $l$ is the number of literal data bytes that follow the length byte. The high order bit of the length byte distinguishes repeated from nonrepeated data: if the high bit is 1, it indicates repeated data, if 0 it indicates literal bytes.

Typical C code to decode PackBits data would be:

```
i = getc( file );          /* Count byte */
if ( i < 0x80 ) {
    {          /* Unpack next (I+1) chars as is */
    for ( j = 0; j <= i; j++ )
```

```
                store( getc( file ) );
        }
    else
        {       /* Repeat next char (2's comp I) times */
        ch = getc( file );
        for ( j = 0; j <= 256 - i; j++ )
                store( ch );
    }
```

PackBits compresses bytes, even though the underlying data may not be byte-oriented. MacPaint files have 1-bit pixels, so in this case PackBits is really compressing 8 pixel chunks of the image. A compression scheme that compressed pixel-by-pixel rather than byte-by-byte would probably provide better compression, but the simplicity of PackBits makes it quite popular.

## 4.2 HUFFMAN CODING

More sophisticated schemes use variable-length codes. Assume that we start with an image consisting of 8-bit pixels. Unless the image is totally random, some pixel values occur far more often than do others. We can assign a set of codes of different lengths to the values, assigning short codes, 4 bits or so, to the most common values, and longer codes, up to 10 or 11 bits, to the less common values. Even allowing for storing a table that maps code values to the original pixel values, this sort of recoding can end up making the file considerably smaller. Variable-length codes are usually called *Huffman* codes, after David Huffman, who figured out in 1952 how to assign codes given the relative frequency of bytes in a file, so that the total coded length was minimized.

The code table used by a Huffman code can either be fixed, with the same table being used for all files, or per-image, with each file containing the table for that file. In graphics files, fixed tables are much more common. Group 3 facsimile uses a fixed Huffman code that effectively compresses the scanned monochrome images found in fax transmissions. JPEG uses a simple fixed Huffman-like code as the final stage in storing image values.

## 4.3 DICTIONARY SYSTEMS

Huffman schemes start with fixed-size inputs and assign variable-length codes. Dictionary schemes do the opposite—start with variable-length inputs and assign fixed-length codes. The dictionary scheme most commonly used in graphics files is known as LZW for Lempel, Ziv, and Welch, the three people who invented it. (Welch, working for Unisys, modified an earlier scheme of Lempel and Ziv and was granted U.S. patent number 4,558,302 on it. Miller and Wegman at IBM, working independently at about the same time, were granted patent number 4,814,746 on the same scheme. Fortunately, neither IBM nor Unisys has tried to enforce either patent in graphics applications, although Unisys has said that they may do so in the future.)

An LZW compressor, unlike Huffman, does not need to compute a code table before compressing the input. It is *adaptive*, meaning that it constructs the dictionary on the fly as it reads the input data. The decoding program can deduce from the sequence of codes in the compressed file what dictionary entries the encoding program used, so the dictionary need not be stored explicitly at all.

At the beginning of a file, an LZW compressor starts with a dictionary containing all of the possible input values in the original file. If it is encoding bytes, the dictionary contains the byte values from 0 to 255. Each time it finds a previously unseen sequence of input values, it adds a new entry to the dictionary.

At the beginning of the file, the encoder emits each dictionary entry number as a 9-bit value (9 bits because the table must contain at least 257 entries, the original 256 plus the first entry added). As the dictionary gets bigger, it increases to 10-bit values when the dictionary exceeds 512 entries, to 11-bit values when the dictionary exceeds 1,024 entries, and so forth. Eventually, the dictionary reaches a maximum size that the encoder can handle (in the range of 4K to 64K entries, depending on the memory available to the encoder), at which point it discards the dictionary constructed to that point and starts constructing a new dictionary.

GIF files all use LZW compression, as do some variants of TIFF files. It turns out that for scanned photographs, LZW is not a very effective compression scheme to use, because digitized images tend not to have long identical repeating sequences of pixels. LZW does work well on computer-generated images, and particularly well on large solid color areas, such as borders placed around a picture in order to increase the size of the image to a size such as $640 \times 480$ that matches a popular screen resolution. Nonetheless, GIF is a quite popular format, LZW and all, because it is well standardized and widely available. See Chapter 17 for more details on LZW compression.

## 4.4 MORE COMPLEX SCHEMES

A few other less popular compression schemes are found in some file formats.

### Arithmetic Coding

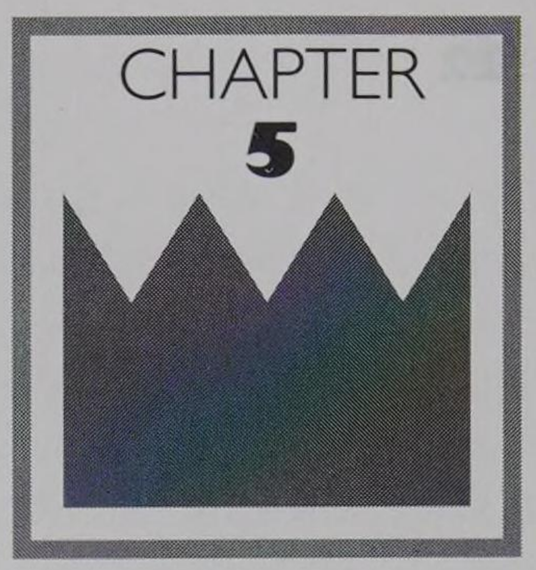
Arithmetic coding is a variable-length coding scheme that works by mapping the set of all possible messages to the unit interval (0,1) and then storing a variable precision binary fraction, representing the point in the interval where the actual message landed. More likely messages are assigned wider chunks of the interval, requiring fewer bits to represent, and less likely messages are assigned narrower chunks needing more bits.

IBM holds several patents on arithmetic compression, and the scheme has not been widely used outside of IBM because of uncertainties about patent licensing.

### Lossy Compression

All of the schemes discussed so far have been *lossless* compressors, in the sense that the decoder can reconstruct the exact same bits that the encoder started with.

Particularly for digitized photographs, small errors in the reconstructed values have little effect on the appearance of the picture. So-called *lossy* schemes take advantage of this fact to achieve compression factors two or three times greater than can be achieved with LZW, while maintaining image quality. The primary lossy technique is JPEG, described in Chapter 19.




# Memory Management for File Processing

One of the knottiest problems in handling graphics files is memory management, since images can contain a lot of data. In this chapter we address some of the memory issues involved in file processing.

## 5.1 HANDLING SMALL AND MEDIUM FILES

Many files are small enough not to need any special processing. Small clip-art images usually are 10K or less, so that no special processing is needed.

For simplicity and consistency in the bitmap code in this book, we use a memory-intensive representation in which bitmap and grayscale images are stored 1 pixel per byte, and color images are stored as 3 bytes per pixel, one each of red, blue, and green. This is usually the fastest representation, since each pixel can be addressed directly by subscripting the pointer to the row of pixels. In many cases, a more compact representation is preferable. For color-mapped images, it is usually at least as convenient to store the indices into the color map rather than the actual colors. (This is particularly so in the common situation when the program will be displaying the image on a screen with a color-mapped frame buffer.)

In the bitmap manipulation code introduced in Parts II through V, pixel data is stored as an array of pointers to rows of pixel data. The array of pointers is dynamically allocated, as are the data rows, which permits straightforward programming even though the size of the arrays is determined at run time. See Figure 5.1.

The pointer to a 2-D pixel array is declared as:

```
pixel **image;
```

(Where pixel is a typedef for the data type holding a pixel's data, usually unsigned char for 8-bit or less grayscale pixels or color indices.) Any particular pixel can be located by straightforward subscripting:

```
int row, column;
pixel mypixel;

mypixel = image[row][column];
```

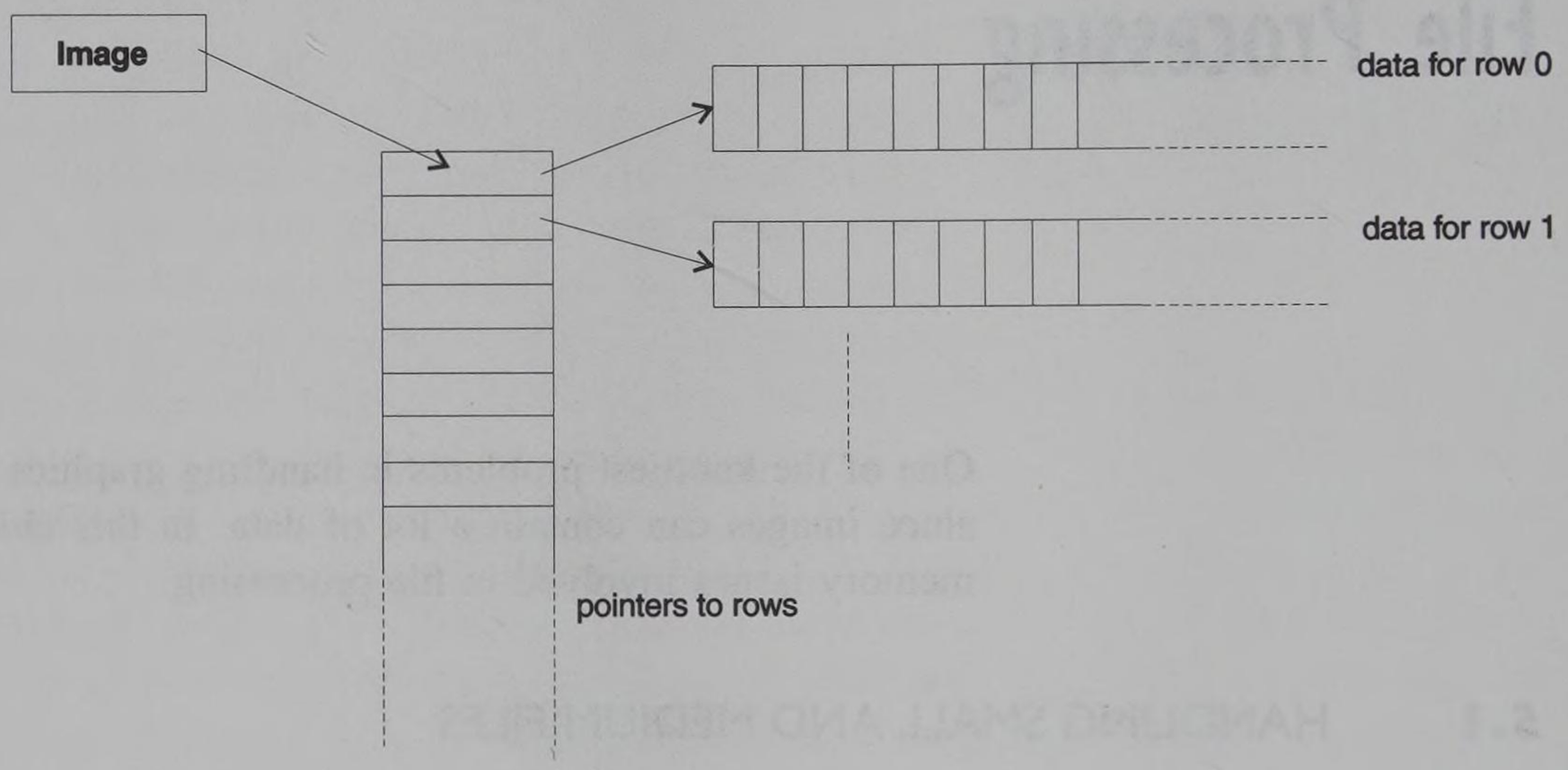



Figure 5.1
Pixel data structure.

When pixels or map indices are less than 8 bits, one can pack them into bytes, saving half of the space with images that are 4 bits per pixel (including most images used under Windows) to as much as seven-eighths of the space for single-bit pixels. The bit packing is most easily handled with C preprocessor macros or C++ in-line functions. For example, for 4-bit pixels:

```
// Get the given 4-bit pixel from the row pointed to by ptr
inline int
getpixel(unsigned char *ptr, int column)
{
    return (column&1)? (ptr[column/2]>>4) : (ptr[column/2]&0xf);
}

// Set the given 4-bit pixel in the row pointed to by ptr
inline void
setpixel(unsigned char *ptr, int column, int val)

{
    unsigned char *pcol = &ptr[column/2]

    if(column&1)
        *pcol = (*pcol&0xf) | (val << 4);
    else
```

```
        *pcol = (*pcol&0x0f) | val;
}
```

## 5.2 HANDLING LARGE FILES IN A VIRTUAL ENVIRONMENT

Virtual memory, long a standard facility on mainframe and medium-sized systems, has become ubiquitous on workstations and is increasingly common on desktop PCs. Desktop Unix systems have had virtual memory for several years, and Windows NT is bringing virtual memory to mainstream desktop PCs.

VM systems largely solve memory handling problems. A 600 by 800 image with 1 byte per pixel, a fairly high-quality grayscale or color-mapped image, fits in 440K of memory, an amount easily handled by most VM systems. Even a 1,024 by 1,280 image with 24 bits per pixel, a high-resolution full-color image, is less than 4MB, which is still not large by VM standards.

Virtual memory isn't a panacea, though. For a program to run, the virtual memory has to be backed up by real memory, and a program that carelessly uses 10 megabytes of virtual memory on a system that only has 4 meg of real memory will run at a snail's pace. Effective use of virtual memory requires that a program cooperate with the working set model that operating systems use to manage virtual memory. In short, VM works by assuming that most programs will need only a fraction of their total address space at any particular time, so only the parts of the program and data that have actually been referenced recently will be resident in memory. Graphics programs should organize memory so that parts of images are stored together that are accessed together. The row-by-row organization we use works well for applications that process a row at a time or that process a few rows at a time. (JPEG compression, for example, processes rows in groups of eight, which should be OK.) Applications that process files a column at a time would be better off with an organization that stored the image by column. Using the row organization, a scan down the column of an image 400 pixels high could in the worst case reference 400 different pages of memory, which would cause poor virtual performance. We discuss some other strategies for lessening memory usage below.

## 5.3 HANDLING LARGE FILES IN AN MS-DOS ENVIRONMENT

In a segmented environment such as MS-DOS, it is often adequate to use large model pointers to rows of pixel data. Since even a very large image is unlikely to be more than a few thousand pixels in height or width, each individual row easily fits within a single segment, as does the pointer array, so this approach works well up to the limits of available memory.

A 640 × 480 image with 8 bits per pixel takes 307K of RAM, which will fit into a 640K system with a moderate-sized program.

## Virtual Segmented Pixel Arrays in C++

MS-DOS systems usually have considerably more than the 640K of memory directly accessible from normal programs. The trick is to get access to it. Over the years a variety of schemes have been designed to allow access to the extra memory, with names like expanded or extended memory, and acronyms like EMS, XMS, VCPI, and DPMI. From the point of view of a DOS program, these schemes all work in roughly the same way: the application calls a service routine to allocate areas of extra memory. Each piece of extra memory is identified by a handle returned from the service routine. When it wants to use a particular piece of extra memory, it calls an *access* routine, passing to it the handle of the piece. The service routine somehow gets the extra memory into regular memory, ideally by adjusting memory mapping hardware, in the worst case by copying it from somewhere else. The access routine then returns the address in regular memory, where the extra memory can now be referenced. The number of pieces of extra memory that can be resident in regular memory varies from as low as one up to several dozen, depending on the hardware involved and the amount of regular memory address space available for the use of the memory manager.

C++ programmers can use operator overloading to create datatypes that can be subscripted just like a regular two-dimensional pointer. This allows conversion of programs that buffer pixel arrays in regular memory with minimal changes.

We define a new class called pixarray that can be used very much like a two-dimensional array of pixels but actually stores each row of pixels in extra memory.

Once a pixarray is allocated, the program references it the same way that it referenced a conventional pixel array. Unlike a regular pixel array, the application doesn't have to worry about deallocating a pixarray, since C++ automatically calls the pixarray's destructor at the end of the block where it is declared.

```
#include "vpixel.h"

void
process_image(int rows, int cols)
{
    int x, y;
    pixarray image(rows, cols); // allocate array
    pixel mypixel;

    image[y][x] = mypixel;      // set arbitrary pixel;
    mypixel = image[y][x];

    // scan a row more efficiently
    pixel *rowptr = image[y];   // get pointer to row
    ...
    mypixel = rowptr[x];        // get a particular pixel
        ...
        // destructor automatically frees the image
}
```

The class also includes routines reading( ) and writing( ) to state whether we are writing pixel data or not (if we're not writing, we don't have to copy rows back to extra memory after we access them) and lockrow( ) and unlockrow( ) routines to force a row to stay in memory. In the absence of locking, a row of pixels is guaranteed only to remain accessible until some other row is referenced. This turns out to be adequate for most but not all of the things that programs do with pixel arrays. A program that needs to reference two or more rows at once has to explicitly lock the rows, do its work, then unlock them.

The implementation of pixarrays is surprisingly simple. This version uses the vmemory package provided by Microsoft C, which provides a consistent interface to the various memory systems; but an implementation using any of the other memory managers would be similar.

The class header file follows:

```
// Skeleton virtual memory pixel array header vpixel.h
// John Levine, 4/93

// Ideally we'd use templates and invent a general array class.
// Lacking templates, we'll do arrays of pixels, since they're the
// bulkiest type we handle.

// build this on the Microsoft vmemory package

#include <vmemory.h>

// full-color 24-bit pixel
typedef unsigned char pixval;
typedef struct
{
    pixval r, g, b;
} pixel;
```

The actual class definition keeps a set of vmemory handles, one per image row. Since the most common access pattern for most images is to look at all the pixels in a row in order, we remember the last row number that we accessed and its in-memory address, so that we can quickly reuse it again next time if the request is for the same row.

The class includes a constructor and destructor, an overloaded [ ] operator that maps in a row, and the other utility routines.

```
class pixarray {
    int nrows, ncols;
    _vmhnd_t *rowhandles;   // each row is a VM handle

    int lastrow;            // last row referenced
    pixel *lastaddr;        // address thereof
```

```
    int dirtyflag;      // are we writing?

    pixarray(const pixarray &x);    // forestall copying
    pixarray operator=(const pixarray &x);
    pixel *deref(const int);
    static int nuses;   // number of arrays allocated
public:
    pixarray(int rows, int cols);
    ~pixarray();
    inline pixel *operator[](const int);
    void lockrow();
    pixel *lockrow(const int);
    void unlockrow();
    void unlockrow(const int);
    void reading();     // references are for reading
    void writing();     // references are for writing
};
```

The inline overloaded subscripting operator checks to see if we are asking for the same row as the previous call. If so, we return the saved pointer. If not, we call the private routine deref( ).

```
inline pixel *
pixarray::operator[](const int row)
{
    // optimize common case of repeated access to the same row
    if(row == lastrow)
        return lastaddr;
    else
        return deref(row);
}
```

We now turn to the implementation code. The vmemory package requires that an application that uses it first initialize the vmemory system with _vheapinit( ), and before exiting clean up with _vheapterm( ). The static class member nuses tracks the number of active pixarrays. In the constructor, we initialize the vmemory system if nuses is 0 before we increment it, and in the destructor we clean up if nuses is 0 after we decrement it. We can lose track of the number of active pixarrays if we allow assignments of pixarrays, so we disallow that by making the copy constructor and assignment operator for pixarray unimplemented private functions. (We should eventually allow for a copy constructor to allow pixarrays to be passed as function arguments, but that would allow multiple pixarrays sharing the same array of row pointers—an extra complication.)

Once the constructor has called _vheapinit( ), if necessary, we make the array rowhandles, which will contain vmemory handles for rows, and call _vmalloc( ) to allocate each row.

We initialize the other local variables to reasonable values and we're done.

```
// Skeleton virtual memory pixel array routine vpixel.cpp
// John Levine, 4/93

#include "vpixel.h"
#include <iostream.h>

const int minparas = 4096;  // minimum size swap area, full segment
int pixarray::nuses = 0;    // first call required initialization

pixarray::pixarray(int rows, int cols)
{
    int i;

    nrows = rows;
    ncols = cols;

if(!nuses) {
    if(!_vheapinit(minparas, _VM_ALLDOS, _VM_ALLSWAP)) {
        cerr << "VM init failed\n";
        rows = 0;
        return;
    }
    cerr << "VM session started";
}
nuses++;

rowhandles = new _vmhnd_t[rows];
if(!rowhandles) {
    cerr << "Pointer array alloc failed\n";
    return;
}

const int rowsize = cols * sizeof(pixel);

for(i = 0; i < rows; i++) {
    _vmhnd_t p = _vmalloc(rowsize);

    if(p == _VM_NULL) {
        cerr << "Can't allocate VM for row " << i << endl;
        return;
    } else
        rowhandles[i] = p;
}
lastrow = -1;
lastaddr = (pixel*)0;
```

```
    dirtyflag = _VM_DIRTY;  // assume we'll be writing
} // pixarray constructor
```

The pixarray destructor undoes what the constructor did. We give back the vmemory with _vfree( ), delete the pointer array, and if this is the last pixarray, terminate the vmemory package.

```
pixarray::~pixarray()
{
    int i;

    for(i = 0; i < nrows; i++)
        _vfree(rowhandles[i]);

    delete []rowhandles;

    if(--nuses <= 0) {
        _vheapterm();
        cerr << "Ending VM session\n";
    }
} // destructor
```

The inline assignment operator calls deref( ) if the trivial case doesn't work. After a sanity check, we call the library _vload( ) to map in the row, remember the pointer it gives us, and return it. We also pass dirtyflag to _vload( ), which tells us whether or not we plan to write on the row.

```
pixel *
pixarray::deref(const int row)
{
        // check subscripts
        if(row < 0 || row >= nrows) {
                cerr << "Invalid vpixel row number " << row << endl;
                return (pixel*)0;
        }
    lastaddr = (pixel*) _vload(rowhandles[row], dirtyflag);
    lastrow = row;
    // cerr << "Loaded row" << row << endl;

    return lastaddr;
} // operator []
```

The row lock and unlock routines just call the corresponding vmemory routines. There are two versions of each, one that locks or unlocks the current row, i.e., the most recently referenced one, and one that takes an explicit row number.

The application declares whether it's planning to read or write with reading( ) and writing( ), each of which sets an appropriate value in dirtyflag.

```
void
pixarray::lockrow()
{
    (void)_vlock(rowhandles[lastrow]);
    // cerr << "Locked row" << lastrow << endl;
}

pixel *
pixarray::lockrow(const int row)
{
    lastaddr = (pixel*)_vlock(rowhandles[lastrow]);
    lastrow = row;
    // cerr << "Locked row" << row << endl;

    return lastaddr;
}

void
pixarray::unlockrow()
{
    _vunlock(rowhandles[lastrow], dirtyflag);
    // cerr << "Unlocked row" << lastrow << endl;
}

void
pixarray::unlockrow(const int row)
{
    _vunlock(rowhandles[row], dirtyflag);
    // cerr << "Unlocked row" << row << endl;
}

void
pixarray::reading()
{
    dirtyflag = _VM_CLEAN;
}

void
pixarray::writing()
{
    dirtyflag = _VM_DIRTY;
}
```

Straightforward techniques like this for using extra memory make it possible to handle large images, up to and (since vmemory will spill to the disk if it has to) beyond the limits of physical memory.

## 5.4 HANDLING PARTIAL FILES

At the cost of some extra complication in one's code, it is often possible to handle a file a chunk at a time. A few file formats, notably TIFF, have explicit support to make it possible. TIFF images are usually logically sliced horizontally and stored as strips (groups of rows) of about 8K bytes apiece, each storing the same number of rows. (The last strip may have fewer rows if the file size isn't an exact multiple of the strip size.) A directory lists the size and location of each strip, so the application can process the strips in any required order. For example, an application that flips the image upside-down could read the strips in reverse order one at a time, never needing more than 1 strip's worth of buffering.

Very large TIFF images are sometimes sliced vertically as well and stored as tiles, or subrectangles, of the overall image. Tiles offer the same advantages as strips for applications that want to process the image by columns rather than by rows. Again, a directory lets the application locate any tile in the file.

Even if a file format doesn't support strips explicitly, it is often possible to simulate them. The application can read through the file, filling an array that notes where each row begins but not actually storing the pixel data. Later processing can consult the array, seek to the appropriate place in the file, and read in the rows as needed. This technique works for any format that compresses each row of pixels independently and for most pixel formats other than GIF and TIFF-LZW (although the latter can use TIFF's explicit stripping or tiling).

For uncompressed formats, such as uncompressed versions of Targa, BMP, and ILBM, it is even easier to handle partial files, since no prescan is required. Once the application has read the file header and discovered the size and depth of the image and the starting position in the file of the pixels, it can simply compute the starting place of a desired row by multiplying the row number by the size in bytes of each row.

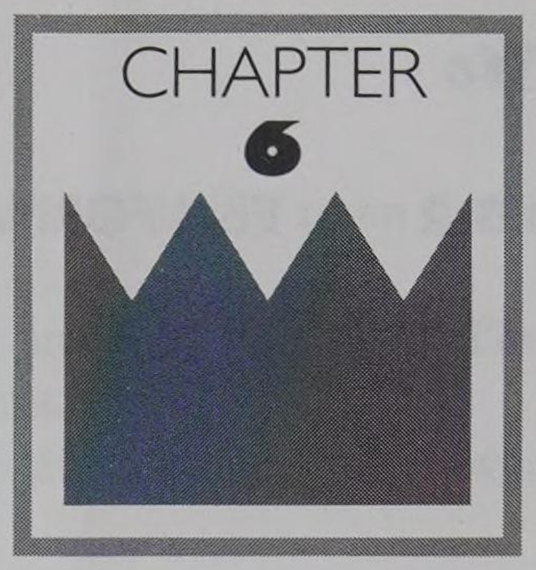
## 5.5 TECHNIQUES FOR VECTOR AND METAFILES

Vector and metafiles require entirely different memory management techniques. The overall size of the file is rarely an issue, since vector and metafile formats are considerably more compact than bitmap formats. Storage difficulties usually arise only when an application is trying to render a file into a bitmap image.

A technique that is quite effective here is *banding*. Rather than try to render the entire image at once, divide the image area into several horizontal bands (similar to the strips discussed above), each of which is small enough to handle conveniently. Then for each band, run through the entire vector or metafile, drawing only the parts that are actually in that band. (In "graphics-ese," clip the rendered image

to the area in the band.) Then once the band is completely rendered, print or otherwise process the pixels and then proceed to the next band. This technique has been used quite successfully in laser printers to avoid having to have enough memory to buffer an entire page image.

# A Framework for Bitmap Files




# A Framework for Bitmap Files

Even though there are dozens of varieties of bitmap files, all of the bitmap formats have far more similarities than differences. All of the formats store a rectangular array of pixel values with some number of bits per pixel. Since the formats are all so similar, we can use a common framework to read and write all kinds of bitmap files.

## 6.1 THE PBM APPROACH

The PBM utilities, which were written mostly by Jef Poskanzer (that's what the "P" stands for) are a group of related programs that read and write a large variety of bitmap file formats and permit a variety of transformations on the files, such as scaling or reducing the number of colors.

PBM defines three extremely simple image formats called PBM, PGM, and PPM, as a "lingua franca" into which all the other formats can be translated. There are also a variety of image filters that read input in one of the PBM formats, do a transformation, and write PBM back out. PBM was originally written on the Unix system, which has *pipe* files that pass directly from one program to another. Using pipes (indicated as a vertical bar), one could read a PCX file, smooth the edges, and produce a GIF file like this:

```
pcxtoppm input.pcx | ppmsmooth | ppmtogif > output.gif
```

The first program, pcxtoppm, reads the PCX file input.pcx and translates it to PPM, one of the PBM formats. Its output is fed to ppmsmooth, which does the smoothing operations and produces a modified PPM file. Its output is fed to ppmtogif, which translates the result to GIF format, which is finally stored in output.gif. On systems without pipes, such as MS-DOS, the intermediate results have to be stored in files, but barring shortages of disk space, the same techniques apply.

## 6.2 PBM FORMATS

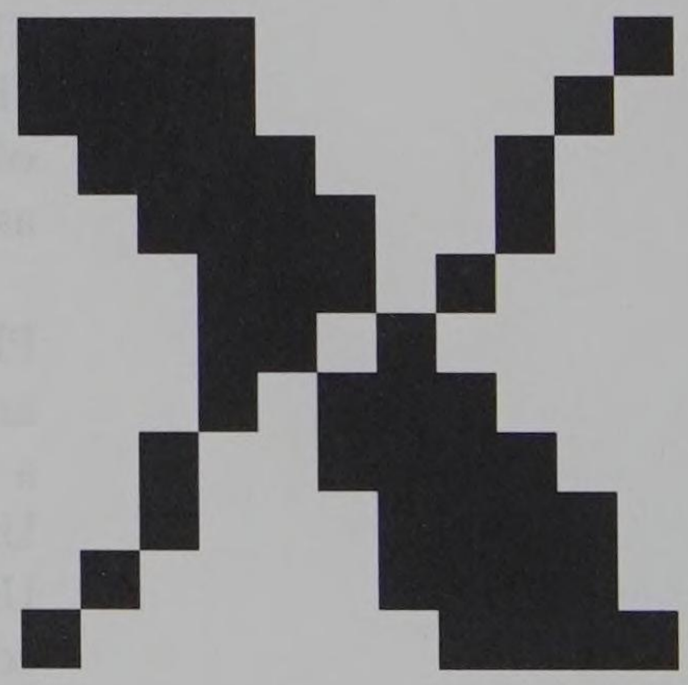
PBM defines six similar file formats. All describe simple rectangular arrays of pixels. Each starts with an ASCII text header consisting of fields separated by white space (spaces and new lines). The header can also contain comment lines that start with a # and run to the next new line character. The first field, the type field, contains the letter P followed by a digit that identifies the particular format. The second and third fields width and height of the image in pixels, as ASCII digits. Some of the subformats have more numeric fields, described below. The header ends with a new line character (decimal 10) after the last header field.

After the header comes the image data, row-by-row, from the top of the image to the bottom. For ASCII formats, pixel data is digit strings separated by spaces. For "raw" formats, pixel data is binary bits or bytes.

### Single-Bit PBM Formats

The simplest format is PBM (Bit Map) for images with 1 bit per pixel. If the type field is P1, the image data is stored as ASCII digits, with a 1 being black and 0 white. If the type field is P4, the data is stored in binary, 8 pixels per byte, with the high bit in each byte being the leftmost pixel. Even if the width of the image is not a multiple of 8 pixels, each row starts at a byte boundary. A typical small PBM file for an $11 \times 11$ pixel image of a stylized X would be:

```
P1
11 11
1 1 1 1 0 0 0 0 0 0 1
1 1 1 1 0 0 0 0 0 1 0
0 1 1 1 1 0 0 0 1 0 0
0 0 1 1 1 1 0 0 1 0 0
0 0 0 1 1 1 0 1 0 0 0
0 0 0 1 1 0 1 0 0 0 0
0 0 0 1 0 1 1 1 0 0 0
0 0 1 0 0 1 1 1 1 0 0
0 0 1 0 0 0 1 1 1 1 0
0 1 0 0 0 0 1 1 1 1 0
1 0 0 0 0 0 0 1 1 1 1
```



### Grayscale PGM Formats

The next simplest format is PGM (Gray Map) format, for grayscale images with a single component more than 1 bit per pixel. The header contains a third number after the image height and width, which is the maximum pixel value. Pixel values can range from 0, meaning black, to the maximum value for white. If the type field is P2, the pixel values are ASCII digit strings, separated by white space. If the type field is P5, the pixel values are bytes, one value per byte. In the P5

format, the largest possible pixel value is 255, since that's all there's room for in a byte. P2 format has no such restriction.

All programs that read PGM files can also recognize and read PBM files. The library code later in this chapter handles the conversion automatically.

A $3 \times 3$ PGM image shading from black at the upper left to white at the lower right might be:

```
P2
3 3
9
0 3 5
3 5 7
5 7 9
```

### Color PPM Formats

The last format is PPM (Pixel Map) format. Each pixel is stored as a triple of red, green, and blue values, in that order. The header contains a third number after the image height and width, which is the maximum component value. If the type field is P3, the pixel values are each three ASCII digit strings, separated by white space. If the type field is P6, the pixel values are triples of bytes, with each byte being a color component. As in the P5 format, the largest possible component value in P6 format is 255. P3 format has no such restriction.

All programs that read PPM files also recognize and read PBM and PGM files and internally convert their contents to pixels where all three components have the same value.

A $4 \times 2$ PPM image with four columns that are black, red, yellow, and white would be:

```
P3
4 2
100
0 0 0   100 0 0   100 100 0   100 100 100
0 0 0   100 0 0   100 100 0   100 100 100
```

### General Formats

Some of the PBM utilities are called PNM (aNy Map) programs. They can read and write any of the PBM, PGM, and PPM formats. Some of the format conversion programs, notably TIFF to PNM, use the PNM libraries because depending on the contents of the TIFF file, the most appropriate format to convert it to might be any of PBM, PGM, or PPM. Other PNM programs perform functions that are equally appropriate to all three formats such as overlaying one image on another. There is no PNM format, just libraries that read and write all of the formats above.

## 6.3 SUPPORT LIBRARIES

All of the bitmap manipulation code that follows is built on top of a set of libraries that handle the various PBM formats. We look in turn at the code to handle PBM, PGM, PPM, and PNM formats, along with general-purpose code to handle all sorts of single-bit, grayscale, full-color, and generic pixel images.

### Header Files

All of the PBM routines include some common header files. The first header file is pbmplus, which contains configuration information used by all programs. The programmer must set an appropriate preprocessor symbol, such as SYSV for Unix System V or MSDOS for MS-DOS, to describe the appropriate system on which the code is running. (These symbols may be set automatically by the compiler or on the command line via a "make" program.)

```
/* pbmplus.h - header file for PBM, PGM, PPM, and PNM
*/

#ifndef _PBMPLUS_H_
#define _PBMPLUS_H_

#include <sys/types.h>
#include <ctype.h>
#include <stdio.h>

#if defined(USG) || defined(SVR4)
#define SYSV
#endif
#if ! ( defined(BSD) || defined(SYSV) || defined(MSDOS) )
/* CONFIGURE: If your system is >= 4.2BSD, set the BSD option;
** if you're a System V site, set the SYSV option;
** and if you're IBM-compatible, set MSDOS.
** If your compiler is ANSI C, you're probably better off
** setting SYSV - all it affects is string handling.
*/
/* #define BSD */
/* #define SYSV */
/* #define MSDOS */
#endif
```

The programmer can set a few configuration parameters to save space in the files and within the program. In practice, one always turns on the PBMPLUS_RAWBITS option because the savings in time and space are so great. The primary reasons one might want an ASCII file are for debugging and if files are to be sent as electronic mail via a system that doesn't support binary attachments. In those cases, as the comments suggest, the program pnmnoraw can force any PBM, PGM, or PPM file into ASCII format.

```
/* CONFIGURE: If you want to enable writing "raw" files, set this
** option. "Raw" files are smaller, and much faster to read and
** write, but you must have a filesystem that allows all 256 ASCII
** characters to be read and written.  You will no longer be able
** to mail P?M files without using uuencode or the equivalent, or
** running the files through pnmnoraw.
** Note that reading "raw" files works whether writing is enabled
** or not.
*/
#define PBMPLUS_RAWBITS
```

Two other space parameters are PGM_BIGGRAYS, which allows grayscale values to range up to 65,535 rather than 255, at a cost of doubling the memory required to process each image; and PPM_PACKCOLORS, which, in connection with PBM_BIGGRAYS, uses a 32-bit per pixel representation for PPM.

The vast majority of images processed on personal computers use 8 bits or less per component, so neither of these options is necessary.

```
/* CONFIGURE: PGM can store gray values as either bytes or shorts.
** For most applications, bytes will be big enough, and the memory
** savings can be substantial.  However, if you need more than
** 8 bits of grayscale resolution, then define this symbol.
*/
/* #define PGM_BIGGRAYS */

/* CONFIGURE: Normally, PPM handles a pixel as a struct of 3 grays.
** If grays are stored in bytes, that's 24 bits per color pixel;
** if grays are stored as shorts, that's 48 bits per color pixel.
** PPM can also be configured to pack the 3 grays into a single
** longword, 10 bits each, 30 bits per pixel.
**
** If you have configured PGM with the PGM_BIGGRAYS option,
** AND you don't need more than 10 bits for each color component,
** AND you care more about memory use than speed,
** then this option might be a win.  Under these circumstances
** it will make some of the programs use 1.5 times less space,
** but all of the programs will run about 1.4 times slower.
**
** If you are not using PGM_BIGGRAYS, then this option is useless
** -- it doesn't save any space, but it still slows things down. */
/* #define PPM_PACKCOLORS */
```

Finally, we have a group of miscellaneous function definitions of routines used by the various programs. Notice the ARGS( ) macro used in function prototypes, which allows a single definition to work either in ANSI C, which supports prototypes, and traditional C, which doesn't.

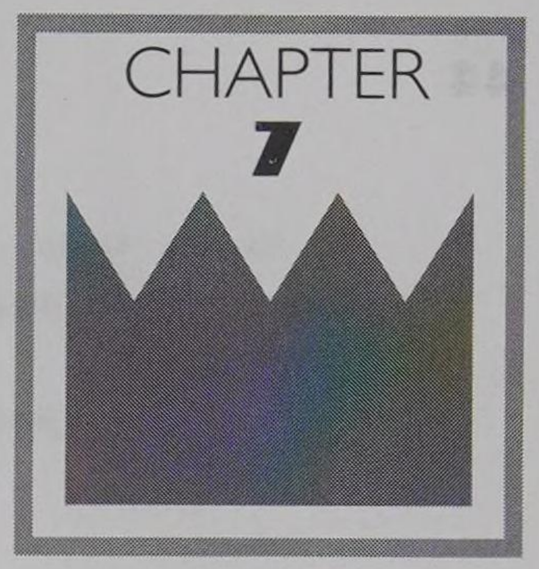
```
/* CONFIGURE: uncomment this to enable debugging checks. */
/* #define DEBUG */

#ifdef SYSV

#include <string.h>
#define index(s,c) strchr(s,c)
#define rindex(s,c) strrchr(s,c)
#define srandom(s) srand(s)
#define random rand
#define bzero(dst,len) memset(dst,0,len)
#define bcopy(src,dst,len) memcpy(dst,src,len)
#define bcmp memcmp
extern void srand();
extern int rand();

#else /*SYSV*/

#include <strings.h>
extern void srandom();
extern long random();

#endif /*SYSV*/

extern int atoi();
extern void exit();
extern long time();
extern int write();

/* CONFIGURE: On some systems, malloc.h doesn't declare these, so
** we have to do it.  On other systems, for example, HP/UX, it
** declares them incompatibly.  And some systems, for example,
** Dynix, don't have a malloc.h  at all.  A sad situation.  If
** you have compilation problems that point here, feel free to
** tweak or remove these declarations.
*/
#include <malloc.h>
extern char* malloc();
extern char* realloc();
extern char* calloc();

/* End of configurable definitions. */

#undef max
#define max(a,b) ((a) > (b) ? (a) : (b))
#undef min
#define min(a,b) ((a) < (b) ? (a) : (b))
#undef abs
#define abs(a) ((a) >= 0 ? (a) : -(a))
```

```
#undef odd
#define odd(n) ((n) & 1)


/* Definitions to make PBMPLUS work with either ANSI C or
C Classic. */

#if __STDC__
#define ARGS(alist) alist
#else /*__STDC__*/
#define ARGS(alist) ()
#define const
#endif /*__STDC__*/


/* Initialization. */

void pm_init ARGS(( int* argcP, char* argv[] ));

/* Variable-sized arrays definitions. */

char** pm_allocarray ARGS(( int cols, int rows, int size ));
char* pm_allocrow ARGS(( int cols, int size ));
void pm_freearray ARGS(( char** its, int rows ));
void pm_freerow ARGS(( char* itrow ));

/* Case-insensitive keyword matcher. */

int pm_keymatch ARGS(( char* str, char* keyword, int minchars ));


/* Log base two hacks. */

int pm_maxvaltobits ARGS(( int maxval ));
int pm_bitstomaxval ARGS(( int bits ));

/* Error handling definitions. */

void pm_message ARGS(( char*, ... ));

void pm_error ARGS(( char*, ... ));         /* doesn't return */
void pm_perror ARGS(( char* reason ));  /* doesn't return */
void pm_usage ARGS(( char* usage ));        /* doesn't return */

/* File open/close that handles "-" as stdin and checks errors.
*/

FILE* pm_openr ARGS(( char* name ));
FILE* pm_openw ARGS(( char* name ));
void pm_close ARGS(( FILE* f ));
```

```
/* Endian I/O. */

int pm_readbigshort ARGS(( FILE* in, short* sP ));
int pm_writebigshort ARGS(( FILE* out, short s ));
int pm_readbiglong ARGS(( FILE* in, long* lP ));
int pm_writebiglong ARGS(( FILE* out, long l ));
int pm_readlittleshort ARGS(( FILE* in, short* sP ));
int pm_writelittleshort ARGS(( FILE* out, short s ));
int pm_readlittlelong ARGS(( FILE* in, long* lP ));
int pm_writelittlelong ARGS(( FILE* out, long l ));

#endif /*_PBMPLUS_H_*/
```

Also used in all programs is the header version.h, which merely defines a date string defining the version of these programs in use.

```
/* version.h - define the current version of PBM, PGM, PPM, and PNM
*/

#define PBMPLUS_VERSION "10dec91"
```




# Libraries for Monochrome Images

All of the PBM (and PGM, PPM, and PNM) programs are built on top of a library of functions that perform basic utilities such as parsing command line arguments, setting up files for I/O, and memory allocation. We turn our attention to those libraries before looking at the programs that use them.

## 7.1 PBM ROUTINES

We look first at the routines used in programs that handle single-bit PBM files. All of these routines are also used, sometimes indirectly, by PGM, PPM, and PNM programs.

PBM files represent rectangular arrays of pixels, with each pixel represented by a single bit, with 0 being white and 1 being black. Such files are quite common, some coming from older image software, such as the ubiquitous MacPaint, and some from scanners, particularly fax attachments to computers.

### PBM Header Files

All PBM, PGM, PPM, and PNM programs include the header pbm.h, which defines basic constants and macros. It starts by defining the values used for black (1) and white (0), the "magic numbers" that start the two kinds of PBM files, "P1" and "P4," and a macro to test whether a particular file's magic number identifies it as either an ASCII or RAW PBM file.

It also defines the type bit, which the libraries use to store each PBM pixel as an unsigned character. This wastes seven-eighth of the space, but is far easier to handle than bits packed into bytes.

```
/* pbm.h - header file for libpbm portable bitmap library
 */

#ifndef _PBM_H_
```

```
#define _PBM_H_

#include "pbmplus.h"

typedef unsigned char bit;
#define PBM_WHITE 0
#define PBM_BLACK 1


/* Magic constants. */

#define PBM_MAGIC1 'P'
#define PBM_MAGIC2 '1'
#define RPBM_MAGIC2 '4'
#define PBM_FORMAT (PBM_MAGIC1 * 256 + PBM_MAGIC2)
#define RPBM_FORMAT (PBM_MAGIC1 * 256 + RPBM_MAGIC2)
#define PBM_TYPE PBM_FORMAT


/* Macro for turning a format number into a type number. */

#define PBM_FORMAT_TYPE(f) ((f) == PBM_FORMAT ||
(f) == RPBM_FORMAT ? PBM_TYPE : -1)
```

Then pbm.h defines the public routines found in the PBM library, notably the memory allocation routines that allocate and free one- and two-dimensional storage arrays. The actual routines have names starting with "pm_." Routines in the PBM libraries and applications call them via names starting with "pbm_," and macros here define the latter in terms of the former. This two-level naming leaves the possibility open that at some point the PBM routines may need to do something PBM-specific, at which point the pbm_ routines could be redefined as needed. As we will see in the next three chapters, the PGM, PPM, and PNM libraries also define storage allocations routines with names starting with "pgm_," "ppm_," and "pnm_," all defined as macros that call the pm_ routines. All of the definitions are the same from format to format except for the call to pm_allocrow( ), in which a parameter varies depending on the amount of storage a pixel takes.

```
/* Declarations of routines. */

void pbm_init ARGS(( int* argcP, char* argv[] ));

#define pbm_allocarray( cols, rows ) ((bit**)
        pm_allocarray( cols, rows, sizeof(bit) ))
#define pbm_allocrow( cols )
((bit*) pm_allocrow( cols, sizeof(bit) ))
#define pbm_freearray( bits, rows )
pm_freearray( (char**) bits, rows )
#define pbm_freerow( bitrow ) pm_freerow( (char*) bitrow )
```

```
bit** pbm_readpbm ARGS(( FILE* file, int* colsP, int* rowsP ));
void pbm_readpbminit ARGS(( FILE* file, int* colsP, int* rowsP,
     int* formatP ));
void pbm_readpbmrow ARGS(( FILE* file, bit* bitrow,
int cols, int format ));

void pbm_writepbm ARGS(( FILE* file, bit** bits, int cols,
     int rows, int forceplain ));
void pbm_writepbminit ARGS(( FILE* file, int cols, int rows,
     int forceplain ));
void pbm_writepbmrow ARGS(( FILE* file, bit* bitrow, int cols,
     int forceplain ));

#endif /* PBM_H */
```

The very short header libpbm.h defines internal routines used within the PBM library. All are defined by code later in this chapter.

```
/* libpbm.h - internal header file for libpbm portable
bitmap library */

#ifndef _LIBPBM_H_
#define _LIBPBM_H_

/* Here are some routines internal to the pbm library. */

char pbm_getc ARGS(( FILE* file ));
unsigned char pbm_getrawbyte ARGS(( FILE* file ));
int pbm_getint ARGS(( FILE* file ));

int pbm_readmagicnumber ARGS(( FILE* file ));

void pbm_readpbminitrest ARGS(( FILE* file, int* colsP,
int* rowsP ));

#endif /*_LIBPBM_H_*/
```

## 7.2 MEMORY ALLOCATION ROUTINES

The first set of library routines supports variably sized rectangular arrays, the rectangular arrays of pixels found in PBM, PGM, and PPM images. Since C doesn't directly support variably sized two-dimensional arrays, we simulate them with an array of pointers. Each entry in the pointer array points to a chunk of space representing one row of the image. Since C uses the same syntax for pointer and for array subscripting, with appropriate declarations application code can write expressions like a [row][col] and get the desired result.

Programs that process their input a row at a time usually use pm_allocrow( ) and pm_freerow( ) to allocate and free space for single rows of the image; those

that handle the entire image at once use pm_allocarray( ) and pm_freearray( ) to allocate and free the whole array. On machines with flat addressing, the array is allocated as one large chunk, and pointers to appropriate places within it are stored into the pointer vector. MS-DOS machines don't support chunks bigger than 64K very well, so each row is allocated separately. This is a little slower than one big allocation, but allows each row to be up to 64K, rather than limiting the whole array to 64K. A possible optimization would be to check the total size of the array and use a single allocation if the entire array fits in 64K or allocate individual rows otherwise. In practice, pm_allocarray( ) is usually called only once or twice per program run, so the performance difference is negligible.

Note that the two allocation routines take a "size" parameter that is the size of each single pixel. The pixel size varies from format to format, so the macros that call the allocation routines, e.g., pbm_allocrow( ) and pbm_allocarray( ), pass the appropriate size for each format.

```
/* libpbm1.c - pbm utility library part 1
*/

#include "pbm.h"
#include "version.h"
#include "libpbm.h"
#if __STDC__
#include <stdarg.h>
#else /*__STDC__*/
#include <varargs.h>
#endif /*__STDC__*/


/* Variable-sized arrays. */

/* allocate a row of cols columns, each of size bytes */
char*
pm_allocrow( cols, size )
    int cols;
    int size;
    {
    register char* itrow;

    itrow = (char*) malloc( cols * size );
    if ( itrow == (char*) 0 )
        pm_error( "out of memory allocating a row" );
    return itrow;
    }

/* free a row allocated by pm_allocrow */
void
pm_freerow( itrow )
    char* itrow;
```

```
    {
    free( itrow );
    }

/* Allocate an array of cols by rows entries,
 * each of the given size
 *
 * MSDOS version: Allocate one row at a time to avoid
 * segment overflow problems
 */
char**
pm_allocarray( cols, rows, size )
    int cols, rows;
    int size;
    {
    char** its;
    int i;

    its = (char**) malloc( rows * sizeof(char*) );
    if ( its == (char**) 0 )
        pm_error( "out of memory allocating an array" );
#ifdef MSDOS
    for ( i = 0; i < rows; ++i )
        its[i] = pm_allocrow( cols, size );
#else /* MSDOS */
    its[0] = (char*) malloc( rows * cols * size );
    if ( its[0] == (char*) 0 )
        pm_error( "out of memory allocating an array" );
    for ( i = 1; i < rows; ++i )
        its[i] = &(its[0][i * cols * size]);
#endif
    return its;
    }

/* free an array allocated by pm_allocarray,
     with the given number of rows */
void
pm_freearray( its, rows )
    char** its;
    int rows;
    {
#ifdef MSDOS
    int i;

    for ( i = 0; i < rows; ++i )
        free(its[i]);
#else /* MSDOS */
    free( its[0] );
#endif /* MSDOS */
```

```
    free( its );
    }
```

## 7.3 PROGRAM STARTUP ROUTINES

The next set of routines supports the startup code in each of the PBM applications. The first, pm_keymatch( ), is a string-matching routine used to match command line arguments. The PBM programs use a verbose Unix style of arguments, e.g., "–reversebits" for an argument that reverses the sense of bits in an image. For the benefit of lazy typists, each argument can be abbreviated to the shortest unambiguous string; the third argument to pm_keymatch( ) gives the number of required characters for a match.

```
/* Case-insensitive keyword matcher. */

/* str: string to match, keyword: literal keyword, minchars: minimum
 * number of characters that have to match
 */
int
pm_keymatch( str, keyword, minchars )
    char* str;
    char* keyword;
    int minchars;
    {
    register int len;

    len = strlen( str );
    if ( len < minchars )
        return 0;
    while ( --len >= 0 )
        {
        register char c1, c2;

        c1 = *str++;
        c2 = *keyword++;
        if ( c2 == '\0' )
            return 0;
        if ( isupper( c1 ) )
            c1 = tolower( c1 );
        if ( isupper( c2 ) )
            c1 = tolower( c2 );
        if ( c1 != c2 )
            return 0;
        }
return 1;
}
```

The routine pm_init( ) is called at the beginning of every PBM, PGM, PPM, and PNM program. It saves the program name and checks for some standard

arguments, "–quiet," which suppresses warning messages, and "–version," which reports the version date and compile options used to build the program.

PBM programs actually call pbm_init( ), which merely calls pm_init( ). It exists as a separate routine in case at some point PBM programs need other initialization.

```
/* Initialization. */

static char* progname;
static int showmessages;

/* initialize the library, argcP is a pointer to the argc value
 * passed to main(), argv is the argument array passed to main
 */
void
pm_init( argcP, argv )
    int* argcP;
    char* argv[];
    {
    int argn, i;

    /* Extract program name. */
    progname = rindex( argv[0], '/');
    if ( progname == NULL )
        progname = argv[0];
    else
        ++progname;

    /* Check for any global args. */
    showmessages = 1;
    for ( argn = 1; argn < *argcP; ++argn )
        {
        if ( pm_keymatch( argv[argn], "-quiet", 6 ) )
            {
            showmessages = 0;
            }
        else if ( pm_keymatch( argv[argn], "-version", 7 ) )
            {
            pm_message( "Version of %s", PBMPLUS_VERSION );
#ifdef BSD
            pm_message( "BSD defined" );
#endif /*BSD*/
#ifdef SYSV
            pm_message( "SYSV defined" );
#endif /*SYSV*/
#ifdef MSDOS
            pm_message( "MSDOS defined" );
#endif /*MSDOS*/
#ifdef PBMPLUS_RAWBITS
            pm_message( "PBMPLUS_RAWBITS defined" );
```

```
#endif /*PBMPLUS_RAWBITS*/
#ifdef PBMPLUS_BROKENPUTC1
            pm_message( "PBMPLUS_BROKENPUTC1 defined" );
#endif /*PBMPLUS_BROKENPUTC1*/
#ifdef PBMPLUS_BROKENPUTC2
            pm_message( "PBMPLUS_BROKENPUTC2 defined" );
#endif /*PBMPLUS_BROKENPUTC2*/
#ifdef PGM_BIGGRAYS
            pm_message( "PGM_BIGGRAYS defined" );
#endif /*PGM_BIGGRAYS*/
#ifdef PPM_PACKCOLORS
            pm_message( "PPM_PACKCOLORS defined" );
#endif /*PPM_PACKCOLORS*/
#ifdef DEBUG
            pm_message( "DEBUG defined" );
#endif /*DEBUG*/
#ifdef NEED_VFPRINTF1
            pm_message( "NEED_VFPRINTF1 defined" );
#endif /*NEED_VFPRINTF1*/
#ifdef NEED_VFPRINTF2
            pm_message( "NEED_VFPRINTF2 defined" );
#endif /*NEED_VFPRINTF2*/
#ifdef RGB_DB
            pm_message( "RGB_DB=\"%s\"", RGB_DB );
#endif /*RGB_DB*/
#ifdef LIBTIFF
            pm_message( "LIBTIFF defined" );
#endif /*LIBTIFF*/
            exit( 0 );
            }
        else
            continue;
        for ( i = argn + 1; i <= *argcP; ++i )
            argv[i - 1] = argv[i];
        --(*argcP);
        }
#ifdef MSDOS
    /* be sure to write standard output as binary */
    _setmode( _fileno(stdout), _O_BINARY);
#endif /* MSDOS */
    } /* pm_init */

/* pbm-specific initialization, same arguments
   as pm_init
*/
void
pbm_init( argcP, argv )
    int* argcP;
    char* argv[];
```

```
{
pm_init( argcP, argv );
}
```

## 7.4 ERROR REPORTING

There is a small but adequate set of error-reporting routines. The routine pm_usage( ) prints a syntax summary and exits, used when a program detects an invalid program argument. When a system call reports an error, pm_perror( ) is an expanded version of the standard perror( ) routine and prints a message identifying both the system call error and optionally some status from the caller.

Warning messages are reported by pm_message( ), which prints the program name and the arguments to pm_message (via fprintf( ), so the arguments to pm_message are of the same form as arguments to printf). It prints nothing if the flag showmessages is 0, meaning that the user specified "–quiet."

Error messages are reported in much the same way by pm_error( ), also with printf( )-like arguments, which then exits. (All errors are considered to be unrecoverable.)

```
/* Error handling. */

void
pm_usage( usage )
    char* usage;
    {
    fprintf( stderr, "usage:  %s %s\n", progname, usage );
    exit( 1 );
    }

void
pm_perror( reason )
    char* reason;
    {
    extern char* sys_errlist[];
    extern int errno;
    char* e;

    e = sys_errlist[errno];

    if ( reason != 0 && reason[0] != '\0' )
        pm_error( "%s - %s", reason, e );
    else
        pm_error( "%s", e );
    }

#if __STDC__
void
pm_message( char* format, ... )
```

```
    {
    va_list args;

    va_start( args, format );
#else /*__STDC__*/
/*VARARGS1*/
void
pm_message( va_alist )
    va_dcl
    { /*}*/
    va_list args;
    char* format;

    va_start( args );
    format = va_arg( args, char* );
#endif /*__STDC__*/

    if ( showmessages )
        {
        fprintf( stderr, "%s: ", progname );
        (void) vfprintf( stderr, format, args );
        fputc( '\n', stderr );
        }
    va_end( args );
    }

#if __STDC__
void
pm_error( char* format, ... )
    {
    va_list args;

    va_start( args, format );
#else /*__STDC__*/
/*VARARGS1*/
void
pm_error( va_alist )
    va_dcl
    { /*}*/
    va_list args;
    char* format;

    va_start( args );
    format = va_arg( args, char* );
#endif /*__STDC__*/

    fprintf( stderr, "%s: ", progname );
    (void) vfprintf( stderr, format, args );
    fputc( '\n', stderr );
```

```
    va_end( args );
    exit( 1 );
    }
```

The source for vfprintf( ) emulation appears at this point in the source files. Since every ANSI C compiler supports vprintf, we've omitted the emulation code.

## 7.5 MISCELLANEOUS SUPPORT

A few routines here perform integer base two logarithms. They are useful to convert from field widths to the maximum values that will fit in the field and vice versa.

```
/* Log base two hacks. */

/* return number of bits needed to represent maxval */
int
pm_maxvaltobits( maxval )
    int maxval;
    {
    if ( maxval <= 1 )
        return 1;
    else if ( maxval <= 3 )
        return 2;
    else if ( maxval <= 7 )
        return 3;
    else if ( maxval <= 15 )
        return 4;
    else if ( maxval <= 31 )
        return 5;
    else if ( maxval <= 63 )
        return 6;
    else if ( maxval <= 127 )
        return 7;
    else if ( maxval <= 255 )
        return 8;
    else if ( maxval <= 511 )
        return 9;
    else if ( maxval <= 1023 )
        return 10;
    else if ( maxval <= 2047 )
        return 11;
    else if ( maxval <= 4095 )
        return 12;
    else if ( maxval <= 8191 )
        return 13;
    else if ( maxval <= 16383 )
```

```
        return 14;
    else if ( maxval <= 32767 )
        return 15;
    else if ( (long) maxval <= 65535L )
        return 16;
    else
        pm_error( "maxval of %d is too large!", maxval );
    }

/* return maximum value that can be represented in bits bits */
int
pm_bitstomaxval( bits )
    int bits;
    {
    return ( 1 << bits ) - 1;
    }
```

## 7.6 FILE HANDLING

PBM programs can read their input either from the standard input or from a file named on the command line. Most programs write their results to the standard output, though a few also write named files. The first file routines open and close files for reading and writing, letting the special case name "-" meaning the standard input, checking for errors, and, on MS-DOS, opening the file in binary mode. (In nonbinary mode, DOS C libraries add and delete carriage return characters before new-line characters, which in this application is never useful and can be disastrous if the carriage return or new line is actually binary image data, not text.)

```
/* File open/close that handles "-" as stdin and checks errors. */

FILE*
pm_openr( name )
    char* name;
    {
    FILE* f;

    if ( strcmp( name, "-" ) == 0 )
        f = stdin;
    else
        {
#ifdef MSDOS
        f = fopen( name, "rb" );
#else /*MSDOS*/
        f = fopen( name, "r" );
#endif /*MSDOS*/
        if ( f == NULL )
            {
```

```
                pm_perror( name );
                exit( 1 );
                }
            }
        return f;
        }

FILE*
pm_openw( name )
    char* name;
    {
    FILE* f;

#ifdef MSDOS
    f = fopen( name, "wb" );
#else /*MSDOS*/
    f = fopen( name, "w" );
#endif /*MSDOS*/
    if ( f == NULL )
        {
        pm_perror( name );
        exit( 1 );
        }
    return f;
    }

void
pm_close( f )
    FILE* f;
    {
    fflush( f );
    if ( ferror( f ) )
        pm_message( "a file read or write error occurred at some
                     point" );
if ( f != stdin )
    if ( fclose( f ) != 0 )
        pm_perror( "fclose" );
}
```

## 7.7 INTEGER READ/WRITE ROUTINES

Many formats include values coded as binary integers. Reading and writing these values is a little tricky for two reasons: the length of the value in the file may not agree with the natural length of an integer on the local machine, and the "byte order" of the the file may not agree with local conventions either. (The two byte orders are known as big-endian and little-endian, after a reference in *Gulliver's Travels*.)

There are routines to read and write short (2-byte) and long (4-byte) variables in either byte order. They all work by reading or writing 1 byte at a time and assembling or disassembling the value with masking and shifting. Note that the reading routines take as an argument a pointer to the place to store the value read, so the routines can return a status code of 0 for a good value read or −1 for error or EOF.

```
/* Endian I/O.
*/

/* read a bigendian short from file in, store in *sP */
int
pm_readbigshort( in, sP )
    FILE* in;
    short* sP;
    {
    int c;

    if ( (c = getc( in )) == EOF )
        return -1;
    *sP = ( c & 0xff ) << 8;
    if ( (c = getc( in )) == EOF )
        return -1;
    *sP |= c & 0xff;
    return 0;
    }

/* write a bigendian short to file out, value to write is s */
#if __STDC__
int
pm_writebigshort( FILE* out, short s )
#else /*__STDC__*/
int
pm_writebigshort( out, s )
    FILE* out;
    short s;
#endif /*__STDC__*/
    {
    (void) putc( ( s >> 8 ) & 0xff, out );
    (void) putc( s & 0xff, out );
    return 0;
    }

/* read a bigendian long from file in, store in *lP */
int
pm_readbiglong( in, lP )
    FILE* in;
    long* lP;
    {
```

```
    int c;

    if ( (c = getc( in )) == EOF )
        return -1;
    *lP = ( c & 0xff ) << 24;
    if ( (c = getc( in )) == EOF )
        return -1;
    *lP |= ( c & 0xff ) << 16;
    if ( (c = getc( in )) == EOF )
        return -1;
    *lP |= ( c & 0xff ) << 8;
    if ( (c = getc( in )) == EOF )
        return -1;
    *lP |= c & 0xff;
    return 0;
    }

/* write a bigendian short to file out, value to write is s */
int
pm_writebiglong( out, l )
    FILE* out;
    long l;
    {
    (void) putc( ( l >> 24 ) & 0xff, out );
    (void) putc( ( l >> 16 ) & 0xff, out );
    (void) putc( ( l >> 8 ) & 0xff, out );
    (void) putc( l & 0xff, out );
    return 0;
    }

/* read little-endian short from file in, store in *sP */
int
pm_readlittleshort( in, sP )
    FILE* in;
    short* sP;
    {
    int c;

    if ( (c = getc( in )) == EOF )
        return -1;
    *sP = c & 0xff;
    if ( (c = getc( in )) == EOF )
        return -1;
    *sP |= ( c & 0xff ) << 8;
    return 0;
    }

/* write a little-endian short to file out, value to write is s */
#if __STDC__
```

```
int
pm_writelittleshort( FILE* out, short s )
#else /*__STDC__*/
int
pm_writelittleshort( out, s )
    FILE* out;
    short s;
#endif /*__STDC__*/
    {
    (void) putc( s & 0xff, out );
    (void) putc( ( s >> 8 ) & 0xff, out );
    return 0;
    }

/* read little-endian long from file in, store in *sP */
int
pm_readlittlelong( in, lP )
    FILE* in;
    long* lP;
    {
    int c;

    if ( (c = getc( in )) == EOF )
        return -1;
    *lP = c & 0xff;
    if ( (c = getc( in )) == EOF )
        return -1;
    *lP |= (long)( c & 0xff ) << 8;
    if ( (c = getc( in )) == EOF )
        return -1;
    *lP |= (long)( c & 0xff ) << 16;
    if ( (c = getc( in )) == EOF )
        return -1;
    *lP |= (long)( c & 0xff ) << 24;
    return 0;
    }

/* write a little-endian long to file out, value to write is s */
int
pm_writelittlelong( out, l )
    FILE* out;
    long l;
    {
    (void) putc( (int)l & 0xff, out );
    (void) putc( (int)( l >> 8 ) & 0xff, out );
    (void) putc( (int)( l >> 16 ) & 0xff, out );
    (void) putc( (int)( l >> 24 ) & 0xff, out );
    return 0;
    }
```

## 7.8 READING PBM FORMAT FILES

Now we turn our attention to reading PBM format files. From the application's point of view, there are two general approaches to reading a file: either it can read the whole file at once, or it can read one row at a time.

To read an entire PBM file, the application calls pbm_readpbm( ), passing it the open stdio file pointer and pointers to variables into which to store the width and height of the image. The return value is a bit** pointer to the effectively two-dimensional array of return values. To read the file, it calls pbm_readpbminit( ), which validates the format and gets the two values for the width and height. Then it allocates the array and reads in the image a row at a time using pbm_readpbmrow( ).

The main task of pbm_readpbminit( ) is to validate the "magic number" at the beginning of the file. It does so by calling pbm_readmagicnumber( ) to read the 2 bytes of magic number. Note that on MS-DOS systems this latter routine has a call to _setmode( ), to be sure there is no text-mode file processing on the file to be read. This routine is always used to read the first two bytes from any PBM, PGM, or PPM file (despite its name, it's used in the PGM, PPM, and PNM libraries as well as in PBM) so it is the logical place to do the _setmode( ).

An application that prefers to save space and process a PBM file a row at a time can directly call pbm_readpbminit( ) to start reading a file, and then call pbm_readpbmrow( ) itself.

```
/* libpbm2.c - pbm utility library part 2 */

#include "pbm.h"
#include "libpbm.h"

/* skip white space and read a 0 or 1 digit for a PBM bit */

/* read a 1 or 0 bit from file */
static bit
pbm_getbit( file )
    FILE* file;
    {
    register char ch;

    do
        {
        ch = pbm_getc( file );
        }
    while ( ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r' );

    if ( ch != '0' && ch != '1' )
        pm_error( "junk in file where bits should be" );

    return ( ch == '1' ) ? 1 : 0;
    }
```

```
/* read and return the two-character magic number from file */

int
pbm_readmagicnumber( file )
    FILE* file;
    {
    int ich1, ich2;

#ifdef MSDOS
    /* be sure to read standard input as binary */
    /* do this here, because it's always the first to read from */
    /* PBM, PGM, or PPM input files */
    if(file == stdin)
        _setmode( _fileno(file), _O_BINARY);
#endif /* MSDOS */
    ich1 = getc( file );
    if ( ich1 == EOF )
        pm_error( "EOF / read error reading magic number" );
    ich2 = getc( file );
    if ( ich2 == EOF )
        pm_error( "EOF / read error reading magic number" );
    return ich1 * 256 + ich2;
    }

/* read the rest of a PBM header, store width and height
in *colsP, *rowsP */

void
pbm_readpbminitrest( file, colsP, rowsP )
    FILE* file;
    int* colsP;
    int* rowsP;
    {
    /* Read size. */
    *colsP = pbm_getint( file );
    *rowsP = pbm_getint( file );
    }

/* read a PBM header, store width and height, and format
 * in *colsP, *rowsP, *formatP
 */
void
pbm_readpbminit( file, colsP, rowsP, formatP )
    FILE* file;
    int* colsP;
    int* rowsP;
    int* formatP;
    {
    /* Check magic number. */
```

```
    *formatP = pbm_readmagicnumber( file );
    switch ( PBM_FORMAT_TYPE(*formatP) )
        {
        case PBM_TYPE:
        pbm_readpbminitrest( file, colsP, rowsP );
        break;

        default:
        pm_error( "bad magic number - not a pbm file" );
        }
    }
```

The distinction between plain and raw PBM files is really made in pbm_readpbmrow( ). For plain files, it calls pbm_getbit( ), shown above, to get a 0 or 1 character signifying the bit. For raw files, it reads literal bytes and unpacks the bits in the raw byte into separate pixels.

```
/* read a PBM row into bitrow[], width cols, given format */
void
pbm_readpbmrow( file, bitrow, cols, format )
    FILE* file;
    bit* bitrow;
    int cols, format;
    {
    register int col, bitshift;
    register unsigned char item;
    register bit* bP;

    switch ( format )
        {
        case PBM_FORMAT:
        for ( col = 0, bP = bitrow; col < cols; ++col, ++bP )
            *bP = pbm_getbit( file );
        break;

    case RPBM_FORMAT:
    bitshift = -1;
    for ( col = 0, bP = bitrow; col < cols; ++col, ++bP )
        {
        if ( bitshift == -1 )
            {
            item = pbm_getrawbyte( file );
            bitshift = 7;
            }
        *bP = ( item >> bitshift ) & 1;
        --bitshift;
        }
    break;
```

```
        default:
        pm_error( "can't happen" );
        }
    }

/*
 * read a PBM file, store width and height, and format
 * in *colsP, *rowsP, *formatP, return pointer to data
 */
bit**
pbm_readpbm( file, colsP, rowsP )
    FILE* file;
    int* colsP;
    int* rowsP;
{
register bit** bits;
int format, row;

pbm_readpbminit( file, colsP, rowsP, &format );

bits = pbm_allocarray( *colsP, *rowsP );

for ( row = 0; row < *rowsP; ++row )
    pbm_readpbmrow( file, bits[row], *colsP, format );

return bits;
}
```

## 7.9 LOW-LEVEL HEADER READING ROUTINES

These routines support the PBM (and later PGM, PPM, and PNM) header reading routines. To read a text byte from a file, they call pbm_getc( ). It acts much like the standard I/O getc( ), except that it checks for EOF and ignores comments by skipping any material from a sharp sign # to the end of a line. For binary bytes, pbm_getrawbyte( ) serves the same function without the comment skipping, since any byte values are allowed in binary data.

```
/* libpbm4.c - pbm utility library part 4
*/

#include "pbm.h"
#include "libpbm.h"

/*
 * read a single text character, check for errors and ignore
 * white space
 */
char
pbm_getc( file )
```

```
FILE* file;
{
register int ich;
register char ch;

ich = getc( file );
if ( ich == EOF )
    pm_error( "EOF / read error" );
ch = (char) ich;

if ( ch == '#' )
    {
    do
        {
        ich = getc( file );
        if ( ich == EOF )
            pm_error( "EOF / read error" );
        ch = (char) ich;
        }
    while ( ch != '\n' && ch != '\r' );
    }

    return ch;
    }

/* read a single byte, check for errors */
unsigned char
pbm_getrawbyte( file )
    FILE* file;
    {
    register int iby;

    iby = getc( file );
    if ( iby == EOF )
        pm_error( "EOF / read error" );
    return (unsigned char) iby;
    }
```

All of the PBM file formats include two or three integers in the header, separated by white space. The routine pbm_getint( ) skips white space and reads an integer value.

```
/* read an integer text value from file */
int
pbm_getint( file )
    FILE* file;
    {
    register char ch;
    register int i;
```

```
do
    {
    ch = pbm_getc( file );
    }
while ( ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r' );

if ( ch < '0' || ch > '9' )
    pm_error( "junk in file where an integer should be" );

i = 0;
do
    {
    i = i * 10 + ch - '0';
    ch = pbm_getc( file );
    }
while ( ch >= '0' && ch <= '9' );

return i;
}
```

## 7.10 WRITING PBM FORMAT FILES

Writing a PBM file is closely analogous to reading. An application can either write the whole file at once by calling pbm_writepbm( ) or else write a row at a time by calling pbm_writepbminit( ), then for each row calling pbm_writepbmrow( ). The various data writing routines all take an argument, forceplain, which makes the library write a plain ASCII file rather than a raw file; in fact none of the programs we will examine writes plain files, since they are eight times as large as raw ones.

To start writing a file, pbm_writepbminit( ) sets binary file mode, if on MSDOS, then writes out the text header.

```
/* libpbm3.c - pbm utility library part 3
*/

#include "pbm.h"
#include "libpbm.h"

/*
 * write PBM header to file, width cols, height rows, raw unless
 * forceplain is set
 */
void
pbm_writepbminit( file, cols, rows, forceplain )
    FILE* file;
    int cols, rows;
    int forceplain;
```

```
    {
#ifdef MSDOS
    /* be sure to write files in binary mode */
    _setmode( _fileno(file), _O_BINARY);
#endif /* MSDOS */
#ifdef PBMPLUS_RAWBITS
    if ( ! forceplain )
        fprintf( file, "%c%c\n%d %d\n",
        PBM_MAGIC1, RPBM_MAGIC2, cols, rows );
    else
        fprintf( file, "%c%c\n%d %d\n",
        PBM_MAGIC1, PBM_MAGIC2, cols, rows );
#else /*PBMPLUS_RAWBITS*/
    fprintf( file, "%c%c\n%d %d\n",
    PBM_MAGIC1, PBM_MAGIC2, cols, rows );
#endif /*PBMPLUS_RAWBITS*/
    }
```

Separate routines write plain and raw files. The only complications are that pbm_writepbmrowraw( ) collects groups of 8 pixels into bytes, and pbm_writepbmrowplain( ) needs to start a new line every 70 characters or so to avoid long lines that can confuse some text editors and mail programs.

```
/* write raw PBM row to file, data in bitrow[], width cols */
#ifdef PBMPLUS_RAWBITS
static void
pbm_writepbmrowraw( file, bitrow, cols )
    FILE* file;
    bit* bitrow;
    int cols;
    {
    register int col, bitshift;
    register unsigned char item;
    register bit* bP;

    bitshift = 7;
    item = 0;
    for ( col = 0, bP = bitrow; col < cols; ++col, ++bP )
        {
        if ( *bP )
            item += 1 << bitshift;
        --bitshift;
        if ( bitshift == -1 )
            {
            (void) putc( item, file );
            bitshift = 7;
            item = 0;
            }
        }
```

```
    if ( bitshift != 7 )
        (void) putc( item, file );
    }
#endif /*PBMPLUS_RAWBITS*/

/* write plain PBM row to file, data in bitrow[],
     width cols */
static void
pbm_writepbmrowplain( file, bitrow, cols )
    FILE* file;
    bit* bitrow;
    int cols;
    {
    register int col, charcount;
    register bit* bP;

    charcount = 0;
    for ( col = 0, bP = bitrow; col < cols; ++col, ++bP )
        {
        if ( charcount >= 70 )
            {
            (void) putc( '\n', file );
            charcount = 0;
            }
        (void) putc( *bP ? '1' : '0', file );
        ++charcount;
        }
    (void) putc( '\n', file );
    }

/* write PBM row to file, data in bitrow[], width cols, raw
 * unless forceplain is set
 */
void
pbm_writepbmrow( file, bitrow, cols, forceplain )
    FILE* file;
    bit* bitrow;
    int cols;
    int forceplain;
    {
#ifdef PBMPLUS_RAWBITS
    if ( ! forceplain )
        pbm_writepbmrowraw( file, bitrow, cols );
    else
        pbm_writepbmrowplain( file, bitrow, cols );
#else /*PBMPLUS_RAWBITS*/
    pbm_writepbmrowplain( file, bitrow, cols );
#endif /*PBMPLUS_RAWBITS*/
    }
```

```
/*
 * write PBM array to file, data in bits[][], width cols, raw
 * format unless forceplain is set
 */
void
pbm_writepbm( file, bits, cols, rows, forceplain )
    FILE* file;
    bit** bits;
    int cols, rows;
    int forceplain;
    {
    int row;

    pbm_writepbminit( file, cols, rows, forceplain );

    for ( row = 0; row < rows; ++row )
        pbm_writepbmrow( file, bits[row], cols, forceplain );
}
```

CHAPTER 8

# Libraries for Grayscale Images

The next level of complexity beyond the single-bit per pixel images that the PBM libraries handle is grayscale. Each pixel, rather than being limited to black and white, can have many shades of gray. The number of shades differs from file to file, with the maximum value stored in the file's header. A 0 pixel value means a black pixel, and the maximum pixel value means white. (This is the opposite of the PBM convention, but is justified by historical conventions.)

Within the PGM code, each pixel is stored in a variable of type "gray." Normally, a gray is an unsigned char, meaning that it can store 256 different values or 256 different gray levels. If the compile-time parameter PGM_BIGGRAYS is set, each gray is stored as an unsigned short value, which increases the number of possible values and gray levels to 65,536. In practice, PGM_BIGGRAYS is infrequently used because the amount of storage required is doubled, and images with more than 256 gray levels are rare.

The PGM file reading routines all understand the PBM file formats and convert PBM files into black-and-white gray values when they read them.

## 8.1 PGM HEADERS

The first header file, pgm.h, is included by PGM, PPM, and PNM programs. This header is considerably smaller than pbm.h, since nearly everything in pbm.h is carried over into PGM programs unchanged.

We declare the pixel type "gray," the constants for a PGM file header (the letters P2 or P5 for plain and raw PGM files), and the macro PGM_FORMAT_TYPE( ), which recognizes the header bytes for the PBM and PGM format files that the PGM routines can read.

```
/* pgm.h - header file for libpgm portable graymap library
*/
```

```
#ifndef _PGM_H_
#define _PGM_H_

#include "pbm.h"

#ifdef PGM_BIGGRAYS
typedef unsigned short gray;
#define PGM_MAXMAXVAL 65535
#else /*PGM_BIGGRAYS*/
typedef unsigned char gray;
#define PGM_MAXMAXVAL 255
#endif /*PGM_BIGGRAYS*/


/* Magic constants. */

#define PGM_MAGIC1 'P'
#define PGM_MAGIC2 '2'
#define RPGM_MAGIC2 '5'
#define PGM_FORMAT (PGM_MAGIC1 * 256 + PGM_MAGIC2)
#define RPGM_FORMAT (PGM_MAGIC1 * 256 + RPGM_MAGIC2)
#define PGM_TYPE PGM_FORMAT


/* Macro for turning a format number into a type number. */

#define PGM_FORMAT_TYPE(f) ((f) == PGM_FORMAT ||
(f) == RPGM_FORMAT ? PGM_TYPE : PBM_FORMAT_TYPE(f))
```

Now we declare the library routines. The storage allocation and freeing routines are macros that call the underlying "pm_" routines we saw in the last chapter. Notice that the definition of pgm_allowrow( ) passes as the third argument the size of a single pixel, "sizeof(gray)," which may be 1 or 2 bytes depending on whether PGM_BIGGRAYS is defined.

```
/* Declarations of routines. */

void pgm_init ARGS(( int* argcP, char* argv[] ));

#define pgm_allocarray( cols, rows )
((gray**) pm_allocarray( cols, rows, sizeof(gray) ))
#define pgm_allocrow( cols ) ((gray*) pm_allocrow(
cols, sizeof(gray) ))
#define pgm_freearray( grays, rows ) pm_freearray(
(char**) grays, rows )
#define pgm_freerow( grayrow ) pm_freerow( (char*) grayrow )

gray** pgm_readpgm ARGS(( FILE* file, int* colsP, int* rowsP,
gray* maxvalP ));
```

```
void pgm_readpgminit ARGS(( FILE* file, int* colsP,
int* rowsP, gray* maxvalP, int* formatP ));

void pgm_readpgmrow ARGS(( FILE* file, gray* grayrow,
int cols, gray maxval, int format ));

void pgm_writepgm ARGS(( FILE* file, gray** grays,
int cols, int rows, gray maxval, int forceplain ));

void pgm_writepgminit ARGS(( FILE* file, int cols,
int rows, gray maxval, int forceplain ));

void pgm_writepgmrow ARGS(( FILE* file, gray* grayrow,
int cols, gray maxval, int forceplain ));

extern gray pgm_pbmmaxval;
/* This is the maxval used when a PGM program reads a
** PBM file.  Normally it is 1; however, for some programs,
** a larger value gives better results
*/

#endif /*_PGM_H_*/
```

The small header file libpgm.h defines an internal routine used in other library routines.

```
/* libpgm.h - internal header file for libpgm portable graymap
** library
*/

#ifndef _LIBPGM_H_
#define _LIBPGM_H_

/* Here are some routines internal to the pgm library. */

void pgm_readpgminitrest ARGS(( FILE* file, int* colsP,
int* rowsP, gray* maxvalP ));

#endif /*_LIBPGM_H_*/
```

## 8.2 PGM INITIALIZATION

All PGM applications start with a call to pgm_init( ). It calls pbm_init( ), which we saw in the last chapter, to process some standard command-line arguments.

```
/* libpgm1.c - pgm utility library part 1
*/
```

```
#include "pgm.h"
#include "libpgm.h"
#include "pbm.h"
#include "libpbm.h"

/* pbm-specific initialization, same arguments as pm_init */
void
pgm_init( argcP, argv )
    int* argcP;
    char* argv[];
    {
    pbm_init( argcP, argv );
    }
```

## 8.3 READING PGM FILES

The code that reads PGM files is, not by coincidence, closely analogous to that used to read PBM files. An application can either call pgm_readpgm( ) to read an entire file at once or else call pgm_readpgminit( ) followed by a call to pgm_readpgmrow( ) for each row in the image.

In either case, the first routine that reads from the file is pgm_readpgminit( ), called either directly or from pgm_readpgm( ). It checks the format type, which must be either PGM or PBM, and calls the appropriate routine to read the rest of the header values. If the file is a PBM file, it prepares to simulate reading a PGM file, using the value in pgm_pbmmaxval, which can be changed by application code, as the maximum gray value, i.e., the value of a white pixel.

```
gray pgm_pbmmaxval = 1;

/*
 * prepare to read PGM or PBM file, store size in *colsP and *rowsP,
 * store max pixel value in *maxvalP, store actual format in
 * *formatP
 */
void
pgm_readpgminit( file, colsP, rowsP, maxvalP, formatP )
    FILE* file;
    int* colsP;
    int* rowsP;
    int* formatP;
    gray* maxvalP;
    {
    /* Check magic number. */
    *formatP = pbm_readmagicnumber( file );
    switch ( PGM_FORMAT_TYPE(*formatP) )
        {
        case PGM_TYPE:
```

```
            pgm_readpgminitrest( file, colsP, rowsP, maxvalP );
            break;

            case PBM_TYPE:
            pbm_readpbminitrest( file, colsP, rowsP );
            *maxvalP = pgm_pbmmaxval;
            break;

            default:
            pm_error( "bad magic number - not a pgm or pbm file" );
            }
        }

/*
 * read PGM hdr values, store size in *colsP and *rowsP,
 * max pixel value in *maxvalP
 */
void
pgm_readpgminitrest( file, colsP, rowsP, maxvalP )
    FILE* file;
    int* colsP;
    int* rowsP;
    gray* maxvalP;
    {
    int maxval;

    /* Read size. */
    *colsP = pbm_getint( file );
    *rowsP = pbm_getint( file );

    /* Read maxval. */
    maxval = pbm_getint( file );
    if ( maxval > PGM_MAXMAXVAL )
        pm_error( "maxval is too large - try reconfiguring with
PGM_BIGGRAYS" );
    *maxvalP = maxval;
    }
```

As each row is read, pgm_readpgmrow( ) takes appropriate action depending on the file format being read. For plain PGM files, it reads ASCII integers using pbm_getint( ), defined in the last chapter. For raw PGM files, it reads binary bytes with pbm_getrawbyte( ), which we also saw in the last chapter. For the two PBM formats, it allocates storage for one row in PBM format, calls the PBM row reading routine, converts the row to PGM, and frees the row. A useful optimization would to be to hold on to the storage for the PBM row, since this routine is usually called in a tight loop.

```
/*
 * Read a row of PGM from file into grayrow[], width cols, max
 * pixel value maxval, file format format
 */
#if __STDC__
void
pgm_readpgmrow( FILE* file, gray* grayrow, int cols, gray maxval,
   int format )
#else /*__STDC__*/
void
pgm_readpgmrow( file, grayrow, cols, maxval, format )
    FILE* file;
    gray* grayrow;
    int cols;
    gray maxval;
    int format;
#endif /*__STDC__*/
    {
    register int col;
    register gray* gP;
    bit* bitrow;
    register bit* bP;

    switch ( format )
        {
        case PGM_FORMAT:
        for ( col = 0, gP = grayrow; col < cols; ++col, ++gP )
            {
            *gP = pbm_getint( file );
#ifdef DEBUG
            if ( *gP > maxval )
                pm_error( "value out of bounds (%u > %u)",
                *gP, maxval );
#endif /*DEBUG*/
            }
        break;

        case RPGM_FORMAT:
        for ( col = 0, gP = grayrow; col < cols; ++col, ++gP )
            {
            *gP = pbm_getrawbyte( file );
#ifdef DEBUG
            if ( *gP > maxval )
                pm_error( "value out of bounds (%u > %u)",
                *gP, maxval );
#endif /*DEBUG*/
            }
        break;
```

```
        case PBM_FORMAT:
        case RPBM_FORMAT:
        bitrow = pbm_allocrow( cols );
        pbm_readpbmrow( file, bitrow, cols, format );
        for ( col = 0, gP = grayrow, bP = bitrow; col < cols;
        ++col, ++gP, ++bP )
            *gP = ( *bP == PBM_WHITE ) ? maxval : 0;
        pbm_freerow( bitrow );
        break;

        default:
        pm_error( "can't happen" );
        }
}
```

Finally, we see pgm_readpgm( ), which puts all of the routines above together.

```
/*
 * Read a whole PGM array from file, store size into *colsP and
 * *rowsP, max pixel value into *maxvalP, return pointer to array
 */
gray**
pgm_readpgm( file, colsP, rowsP, maxvalP )
    FILE* file;
    int* colsP;
    int* rowsP;
    gray* maxvalP;
    {
    gray** grays;
    int row;
    int format;

    pgm_readpgminit( file, colsP, rowsP, maxvalP, &format );

    grays = pgm_allocarray( *colsP, *rowsP );

    for ( row = 0; row < *rowsP; ++row )
        pgm_readpgmrow( file, grays[row], *colsP, *maxvalP,
        format );

    return grays;
    }
```

## 8.4 WRITING PGM FILES

Writing PGM format files is easier than reading them, since there is no format conversion to be done—PGM programs always write PGM files.

The routines are completely parallel to those in the PBM library. Again, an application can either write an entire file by calling pgm_writepgm( ) or write

the file row by row by calling pgm_writepgminit( ) followed by calling pgm_writepgmrow( ) for each row.

The argument "forceplain" to some of the routines makes them write a plain ASCII file rather than raw. In fact, none of the PGM programs uses forceplain, since raw files are smaller and faster to read and write.

```
/* libpgm2.c - pgm utility library part 2
*/

#include "pgm.h"
#include "libpgm.h"

/*
 * Start writing PGM file, size cols x rows, max pixel value maxval,
 * raw unless maxval too big or forceplain set
 */
#if __STDC__
void
pgm_writepgminit( FILE* file, int cols, int rows, gray maxval,
int forceplain )
#else /*__STDC__*/
void
pgm_writepgminit( file, cols, rows, maxval, forceplain )
    FILE* file;
    int cols, rows;
    gray maxval;
    int forceplain;
#endif /*__STDC__*/
    {
#ifdef MSDOS
    /* be sure to write files in binary mode */
    _setmode( _fileno(file), _O_BINARY);
#endif /* MSDOS */
#ifdef PBMPLUS_RAWBITS
    if ( maxval <= 255 && ! forceplain )
        fprintf(
            file, "%c%c\n%d %d\n%d\n", PGM_MAGIC1, RPGM_MAGIC2,
            cols, rows, maxval );
    else
        fprintf(
            file, "%c%c\n%d %d\n%d\n", PGM_MAGIC1, PGM_MAGIC2,
            cols, rows, maxval );
#else /*PBMPLUS_RAWBITS*/
    fprintf(
        file, "%c%c\n%d %d\n%d\n", PGM_MAGIC1, PGM_MAGIC2,
        cols, rows, maxval );
#endif /*PBMPLUS_RAWBITS*/
    }
```

Writing the rows for plain files is slightly more complex than for PBM files since each pixel may be a string of digits rather than a single 1 or 0. The auxiliary routine putus( ) writes a single value as an ASCII string.

```
static void
putus( n, file )
    unsigned short n;
    FILE* file;
    {
    if ( n >= 10 )
        putus( n / 10, file );
    (void) putc( n % 10 + '0', file );
    }
```

```
/* write a row to a plain PGM file from grayrow[],
** width cols */
static void
pgm_writepgmrowplain( file, grayrow, cols, maxval )
    FILE* file;
    gray* grayrow;
    int cols;
    gray maxval;
    {
    register int col, charcount;
    register gray* gP;

    charcount = 0;
    for ( col = 0, gP = grayrow; col < cols; ++col, ++gP )
        {
        if ( charcount >= 65 )
            {
            (void) putc( '\n', file );
            charcount = 0;
            }
        else if ( charcount > 0 )
            {
            (void) putc( ' ', file );
            ++charcount;
            }
#ifdef DEBUG
        if ( *gP > maxval )
        pm_error( "value out of bounds (%u > %u)", *gP, maxval );
#endif /*DEBUG*/
        putus( (unsigned long) *gP, file );
        charcount += 3;
        }
    if ( charcount > 0 )
        (void) putc( '\n', file );
    }
```

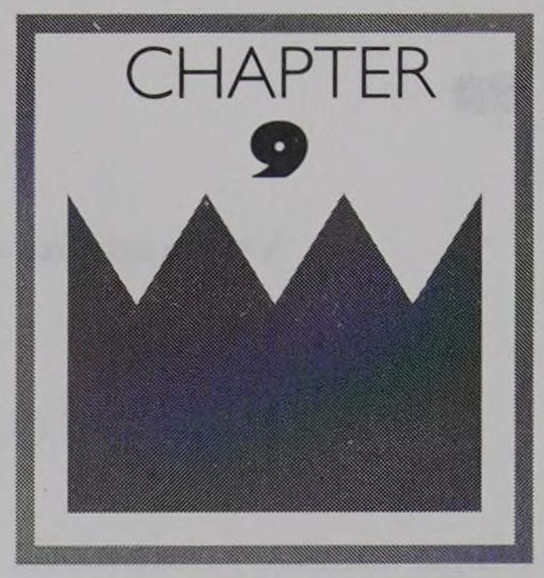
Writing raw PGM files is easier than writing raw PBM files, since no bit packing is required. Each pixel is 1 byte.

```
/* write a row to a raw PGM file from grayrow[], width cols */
#ifdef PBMPLUS_RAWBITS
static void
pgm_writepgmrowraw( file, grayrow, cols, maxval )
    FILE* file;
    gray* grayrow;
    int cols;
    gray maxval;
    {
    register int col;
    register gray* gP;

    for ( col = 0, gP = grayrow; col < cols; ++col, ++gP )
        {
#ifdef DEBUG
        if ( *gP > maxval )
            pm_error( "value out of bounds (%u > %u)",
            *gP, maxval );
#endif /*DEBUG*/
        (void) putc( *gP, file );
        }
    }
#endif /*PBMPLUS_RAWBITS*/
```

Finally, pgm_writepgmrow( ) and pgm_writepgm( ) put all of the routines above together to write rows and entire files.

```
/*
 * write a row to a PGM file from grayrow[], width cols, raw
 * unless maxval too big or forceplain set
 */
#if __STDC__
void
pgm_writepgmrow( FILE* file, gray* grayrow,
int cols, gray maxval, int forceplain )
#else /*__STDC__*/
void
pgm_writepgmrow( file, grayrow, cols, maxval, forceplain )
    FILE* file;
    gray* grayrow;
    int cols;
    gray maxval;
    int forceplain;
#endif /*__STDC__*/
    {
#ifdef PBMPLUS_RAWBITS
```

```
    if ( maxval <= 255 && ! forceplain )
        pgm_writepgmrowraw( file, grayrow, cols, maxval );
    else
        pgm_writepgmrowplain( file, grayrow, cols, maxval );
#else /*PBMPLUS_RAWBITS*/
    pgm_writepgmrowplain( file, grayrow, cols, maxval );
#endif /*PBMPLUS_RAWBITS*/
    }

/*
 * write PGM file from grays[][], size cols by rows, max pixel val
 * maxval, plain unless maxval too big or forceplain set
 */
#if __STDC__
void
pgm_writepgm( FILE* file, gray** grays, int cols,
int rows, gray maxval, int forceplain )
#else /*__STDC__*/
void
pgm_writepgm( file, grays, cols, rows, maxval, forceplain )
    FILE* file;
    gray** grays;
    int cols, rows;
    gray maxval;
    int forceplain;
#endif /*__STDC__*/
    {
    int row;

    pgm_writepgminit( file, cols, rows, maxval, forceplain );

    for ( row = 0; row < rows; ++row )
        pgm_writepgmrow( file, grays[row], cols, maxval,
        forceplain );
    }
```




# Libraries for Full-Color Images

The third group of libraries are for full-color RGB images. Each pixel is stored as three components, one each for red, green, and blue. As with PGM, the maximum value is a parameter stored in the file. The value (0, 0, 0) is black, and (max, max, max) is white.

The PPM file reading routines can also read PBM and PGM files and turn them into PPM data. A gray value v turns into the pixel (v,v,v) with equal red, blue, and green components. A PBM black pixel turns into (0,0,0) and a white one into (m,m,m) where m is the maximum pixel value stored in the library variable ppm_pbmmaxval. It normally contains 1 but can be changed by applications.

Each pixel is stored in a variable of type "pixel," with each component being a "pixval," the same type as a "gray." There are three possible actual representations of pixels and pixvals. If PGM has been compiled without PGM_BIGGRAYS, then a pixval is an unsigned char and a pixel is a 3-byte structure containing the 3 pixvals. If PGM_BIGGRAYS is set, there are two possible pixel formats. The usual one is a 6-byte structure of three 2-byte grays. If PPM_PACKCOLORS is set, each pixel is a 4-byte long value containing three 10-bit components, a space savings of one third.

The header ppm.h defines the type pixval for a component and pixel for a three-component pixel. Since a pixel may be a structure or a long, there are a set of access functions to set and read a pixel. The macro PPM_ASSIGN(p, red, green, blue) assigns the three component values to the pixel p. The macros PPM_GETR(p), PPM_GETG(p), and PPM_GETB(p) extract the red, green, and blue components from the pixel p. The macro PPM_EQUAL(p, q) compares the pixels p and q to see if they are the same color. Although not strictly necessary, this macro speeds up comparisons greatly on packed color pixels.

```
/* ppm.h - header file for libppm portable pixmap library
*/
```

```
#ifndef _PPM_H_
#define _PPM_H_

#include "pgm.h"

typedef gray pixval;

#ifdef PPM_PACKCOLORS

#define PPM_MAXMAXVAL 1023
typedef unsigned long pixel;
#define PPM_GETR(p) (((p) & 0x3ff00000) >> 20)
#define PPM_GETG(p) (((p) & 0xffc00) >> 10)
#define PPM_GETB(p) ((p) & 0x3ff)
#define PPM_ASSIGN(p,red,grn,blu) (p) = ((pixel) (red)
  << 20) | ((pixel) (grn) << 10) | (pixel) (blu)
#define PPM_EQUAL(p,q) ((p) == (q))

#else /*PPM_PACKCOLORS*/

#define PPM_MAXMAXVAL PGM_MAXMAXVAL
typedef struct
    {
    pixval r, g, b;
    } pixel;
#define PPM_GETR(p) ((p).r)
#define PPM_GETG(p) ((p).g)
#define PPM_GETB(p) ((p).b)
#define PPM_ASSIGN(p,red,grn,blu)  do { (p).r = (red);
   (p).g = (grn); (p).b = (blu); } while ( 0 )
#define PPM_EQUAL(p,q) ( (p).r == (q).r && (p).g ==
   (q).g && (p).b == (q).b )

#endif /*PPM_PACKCOLORS*/
```

We define the header constants for plain and raw PPM file magic numbers (P3 and P6, respectively), the type code for PPM files, and the macro PPM_FORMAT_TYPE( ), which recognizes the header constants.

```
/* Magic constants. */

#define PPM_MAGIC1 'P'
#define PPM_MAGIC2 '3'
#define RPPM_MAGIC2 '6'
#define PPM_FORMAT (PPM_MAGIC1 * 256 + PPM_MAGIC2)
#define RPPM_FORMAT (PPM_MAGIC1 * 256 + RPPM_MAGIC2)
#define PPM_TYPE PPM_FORMAT
```

```
/* Macro for turning a format number into a type number. */

#define PPM_FORMAT_TYPE(f) ((f) == PPM_FORMAT ||
(f) == RPPM_FORMAT ? PPM_TYPE : PGM_FORMAT_TYPE(f))
```

Now we declare the public PPM routines. The storage allocation routines again are macros that call the pm_ routines we saw in Chapter 7.

```
/* Declarations of routines. */

void ppm_init ARGS(( int* argcP, char* argv[] ));

#define ppm_allocarray( cols, rows ) \
        ((pixel**) pm_allocarray( cols, rows, sizeof(pixel) ))
#define ppm_allocrow( cols ) \
        ((pixel*) pm_allocrow( cols, sizeof(pixel) ))
#define ppm_freearray( pixels, rows ) \
        pm_freearray( (char**) pixels, rows )
#define ppm_freerow( pixelrow ) pm_freerow( (char*) pixelrow )

pixel** ppm_readppm ARGS(( FILE* file, int* colsP,
int* rowsP, pixval* maxvalP ));
void ppm_readppminit ARGS(( FILE* file, int* colsP,
int* rowsP, pixval* maxvalP, int* formatP ));
void ppm_readppmrow ARGS(( FILE* file, pixel* pixelrow,
int cols, pixval maxval, int format ));

void ppm_writeppm ARGS(( FILE* file, pixel** pixels,
int cols, int rows, pixval maxval, int forceplain ));
void ppm_writeppminit ARGS(( FILE* file, int cols, int rows,
pixval maxval, int forceplain ));
void ppm_writeppmrow ARGS(( FILE* file, pixel* pixelrow,
int cols, pixval maxval, int forceplain ));

pixel ppm_parsecolor ARGS(( char* colorname, pixval maxval ));
char* ppm_colorname ARGS(( pixel* colorP, pixval maxval,
int hexok ));

extern pixval ppm_pbmmaxval;
/* This is the maxval used when a PPM program reads a PBM file.
** Normally it is 1; however, for some programs, a larger value
** gives better results
*/
```

Finally, we define two more macros useful in writing pixel manipulation routines. The first is PPM_DEPTH(np, op, omax, nmax), which scales a pixel to a new maximum component value. It assigns the contents of pixel op, with maximum values omax to pixel np, with maximum values nmax.

PPM_LUMIN(p) computes the "luminance" of a pixel, which is, roughly speaking, its brightness. The luminance of a black pixel is 0, of a white pixel is maxval, and of anything else some value in between. The luminance of a pixel is a weighted average of the three components, with different weights to account for the fact that the human eye is more sensitive to some colors than to others. (The weighting factors used are fairly standard ones from the television industry.)

```
/* Color scaling macro -- to make writing ppmtowhatever easier. */

#define PPM_DEPTH(newp,p,oldmaxval,newmaxval) \
    PPM_ASSIGN( (newp), \
        ( (int) PPM_GETR(p) * (newmaxval) \
              + (oldmaxval) / 2 ) / (oldmaxval), \
        ( (int) PPM_GETG(p) * (newmaxval) + (oldmaxval) / 2 ) \
              / (oldmaxval), \
       ( (int) PPM_GETB(p) * (newmaxval) + (oldmaxval) / 2 )
              / (oldmaxval) )

/* Luminance macro. */

#define PPM_LUMIN(p) ( 0.299 * PPM_GETR(p) + 0.587 \
       * PPM_GETG(p) + 0.114 * PPM_GETB(p) )

#endif /*_PPM_H_*/
```

The file libppm.h defines one internal library routine used in other libraries.

```
/* libppm.h - internal header file for libppm portable
** pixmap library
*/

#ifndef _LIBPPM_H_
#define _LIBPPM_H_

/* Here are some routines internal to the ppm library. */

void ppm_readppminitrest ARGS(( FILE* file, int* colsP,
int* rowsP, pixval* maxvalP ));

#endif /*_LIBPPM_H_*/
```

## 9.1 PPM INITIALIZATION

All PPM applications start with a call to ppm_init( ). It calls pbm_init( ), which we saw in Chapter 7, to processes some standard command-line arguments.

```
/* libppm1.c - ppm utility library part 1
*/
```

```
#include "ppm.h"
#include "libppm.h"
#include "pgm.h"
#include "libpgm.h"
#include "pbm.h"
#include "libpbm.h"

/* ppm-specific initialization, same arguments as pm_init */
void
ppm_init( argcP, argv )
    int* argcP;
    char* argv[];
    {
    pgm_init( argcP, argv );
    }
```

## 9.2 READING PPM FILES

The PPM reading code is quite similar to the PGM and PBM reading code. An application can either call ppm_readppm( ) to read an entire file at once or else call ppm_readppminit( ) followed by a call to ppm_readppmrow( ) for each row in the image.

In either case, the first routine that reads from the file is ppm_readppminit( ), called either directly or from ppm_readppm( ). It checks the format type, which may be PPM, PGM, or PBM, and calls the appropriate routine to read the rest of the header values. If the file is a PBM file, it prepares to simulate reading a PPM file, using the value in ppm_pbmmaxval, which can be changed by application code, as the maximum component value. If the file is a PGM file, it uses the maximum gray value as the maximum pixel value.

```
pixval ppm_pbmmaxval = 1;

/*
 * prepare to read PPM, PGM or PBM file, store size in *colsP
 * and *rowsP, store max pixel value in *maxvalP, store actual
 * format in *formatP
 */
void
ppm_readppminit( file, colsP, rowsP, maxvalP, formatP )
    FILE* file;
    int* colsP;
    int* rowsP;
    int* formatP;
    pixval* maxvalP;
    {
    /* Check magic number. */
    *formatP = pbm_readmagicnumber( file );
```

```
switch ( PPM_FORMAT_TYPE(*formatP) )
    {
    case PPM_TYPE:
    ppm_readppminitrest( file, colsP, rowsP, maxvalP );
    break;

    case PGM_TYPE:
    pgm_readpgminitrest( file, colsP, rowsP, maxvalP );
    break;

    case PBM_TYPE:
    pbm_readpbminitrest( file, colsP, rowsP );
    *maxvalP = ppm_pbmmaxval;
    break;

        default:
        pm_error( "bad magic number - not a ppm, pgm,
        or pbm file" );
        }
    }

/*
 * read PPM hdr values, store size in *colsP and *rowsP,
 * max component value in *maxvalP
 */
void
ppm_readppminitrest( file, colsP, rowsP, maxvalP )
    FILE* file;
    int* colsP;
    int* rowsP;
    pixval* maxvalP;
    {
    int maxval;

    /* Read size. */
    *colsP = pbm_getint( file );
    *rowsP = pbm_getint( file );

    /* Read maxval. */
    maxval = pbm_getint( file );
    if ( maxval > PPM_MAXMAXVAL )
        pm_error(
"maxval is too large - try reconfiguring with PGM_BIGGRAYS\n
or without PPM_PACKCOLORS" );
    *maxvalP = maxval;
    }
```

As each row is read, ppm_readppmrow( ) takes appropriate action depending on the file format being read. For plain PPM files, it reads triplets of integers representing the red, green, and blue components using pbm_getint( ), defined in Chapter 7. For raw ppm files, it reads binary bytes with pbm_getrawbyte( ), which we saw in Chapter 8.

For files in PGM and PBM formats, it allocates storage for one row in the file's native format, calls the native row reading routine, converts the row to PPM, and frees the row. A useful optimization would to be to hold on to the storage for the PBM or PGM row, since this routine is usually called in a tight loop.

```
/*
 * Read a row of PPM from file into pixelrow[], width cols, max
 * component value maxval, file format format
 */
#if __STDC__
void
ppm_readppmrow( FILE* file, pixel* pixelrow, int cols,
pixval maxval, int format )
#else /*__STDC__*/
void
ppm_readppmrow( file, pixelrow, cols, maxval, format )
    FILE* file;
    pixel* pixelrow;
    int cols, format;
    pixval maxval;
#endif /*__STDC__*/
    {
    register int col;
    register pixel* pP;
    register pixval r, g, b;
    gray* grayrow;
    register gray* gP;
    bit* bitrow;
    register bit* bP;

    switch ( format )
        {
        case PPM_FORMAT:
        for ( col = 0, pP = pixelrow; col < cols; ++col, ++pP )
            {
            r = pbm_getint( file );
#ifdef DEBUG
            if ( r > maxval )
                pm_error( "r value out of bounds (%u > %u)", r,
                maxval );
#endif /*DEBUG*/
            g = pbm_getint( file );
#ifdef DEBUG
            if ( g > maxval )
```

```
                    pm_error( "g value out of bounds (%u > %u)", g,
                    maxval );
#endif /*DEBUG*/
                b = pbm_getint( file );
#ifdef DEBUG
                if ( b > maxval )
                    pm_error( "b value out of bounds (%u > %u)", b,
                    maxval );
#endif /*DEBUG*/
                PPM_ASSIGN( *pP, r, g, b );
                }
            break;

            case RPPM_FORMAT:
            for ( col = 0, pP = pixelrow; col < cols; ++col, ++pP )
                {
                r = pbm_getrawbyte( file );
#ifdef DEBUG
                if ( r > maxval )
                    pm_error( "r value out of bounds (%u > %u)", r,
                    maxval );
#endif /*DEBUG*/
                g = pbm_getrawbyte( file );
#ifdef DEBUG
                if ( g > maxval )
                    pm_error( "g value out of bounds (%u > %u)", g,
                    maxval );
#endif /*DEBUG*/
                b = pbm_getrawbyte( file );
#ifdef DEBUG
                if ( b > maxval )
                    pm_error( "b value out of bounds (%u > %u)",
                    b, maxval );
#endif /*DEBUG*/
                PPM_ASSIGN( *pP, r, g, b );
                }
            break;

            case PGM_FORMAT:
            case RPGM_FORMAT:
            grayrow = pgm_allocrow( cols );
            pgm_readpgmrow( file, grayrow, cols, maxval, format );
            for ( col = 0, gP = grayrow, pP = pixelrow;
              col < cols; ++col, ++gP, ++pP )
                {
                r = *gP;
                PPM_ASSIGN( *pP, r, r, r );
                }
            pgm_freerow( grayrow );
```

```
        break;

    case PBM_FORMAT:
    case RPBM_FORMAT:
    bitrow = pbm_allocrow( cols );
    pbm_readpbmrow( file, bitrow, cols, format );
    for ( col = 0, bP = bitrow, pP = pixelrow;
    col < cols; ++col, ++bP, ++pP )
        {
        r = ( *bP == PBM_WHITE ) ? maxval : 0;
        PPM_ASSIGN( *pP, r, r, r );
        }
    pbm_freerow( bitrow );
    break;

        default:
        pm_error( "can't happen" );
        }
    }
```

Finally, we see ppm_readppm( ), which puts all of the routines above together.

```
/*
 * Read a whole PPM array from file, store size into *colsP and
 * *rowsP, max component value into *maxvalP, return pointer to
 * array
 */
pixel**
ppm_readppm( file, colsP, rowsP, maxvalP )
    FILE* file;
    int* colsP;
    int* rowsP;
    pixval* maxvalP;
    {
    pixel** pixels;
    int row;
    int format;

    ppm_readppminit( file, colsP, rowsP, maxvalP, &format );

    pixels = ppm_allocarray( *colsP, *rowsP );

    for ( row = 0; row < *rowsP; ++row )
        ppm_readppmrow( file, pixels[row], *colsP, *maxvalP,
        format );

    return pixels;
    }
```

## 9.3 WRITING PPM FILES

Writing PPM, like writing PGM format files, is easier than reading them, since again there is no format conversion to be done—PPM programs always write PPM files.

The routines are completely parallel to those in the PBM and PGM libraries. An application can either write an entire file by calling ppm_writeppm( ) or by writing the file row-by-row by calling ppm_writeppminit( ) followed by calling ppm_writeppmrow( ) for each row.

The argument "forceplain" is interpreted as it is by the PGM routines and, again, none of the PPM programs uses forceplain.

```
/* libppm2.c - ppm utility library part 2
*/

#include "ppm.h"
#include "libppm.h"

/*
 * Start writing PPM file, size cols x rows, max component
 * value maxval, raw unless maxval too big or forceplain set
 */
#if __STDC__
void
ppm_writeppminit( FILE* file, int cols, int rows, pixval maxval,
int forceplain )
#else /*__STDC__*/
void
ppm_writeppminit( file, cols, rows, maxval, forceplain )
    FILE* file;
    int cols, rows;
    pixval maxval;
    int forceplain;
#endif /*__STDC__*/
    {
#ifdef MSDOS
    /* be sure to write files in binary mode */
    _setmode( _fileno(file), _O_BINARY);
#endif /* MSDOS */
#ifdef PBMPLUS_RAWBITS
    if ( maxval <= 255 && ! forceplain )
        fprintf(
            file, "%c%c\n%d %d\n%d\n", PPM_MAGIC1, RPPM_MAGIC2,
            cols, rows, maxval );
    else
        fprintf(
            file, "%c%c\n%d %d\n%d\n", PPM_MAGIC1, PPM_MAGIC2,
            cols, rows, maxval );
#else /*PBMPLUS_RAWBITS*/
```

```
    fprintf(
        file, "%c%c\n%d %d\n%d\n", PPM_MAGIC1, PPM_MAGIC2,
        cols, rows, maxval );
#endif /*PBMPLUS_RAWBITS*/
    }
```

Writing the rows for plain files is slightly more complex than for the previous formats, since each pixel is 3 strings of digits. Another copy of the auxiliary routine putus( ) writes a single value as an ASCII string.

Writing raw PPM files involves writing 3 bytes per pixel, one per component.

```
static void
putus( n, file )
    unsigned short n;
    FILE* file;
    {
    if ( n >= 10 )
        putus( n / 10, file );
    (void) putc( n % 10 + '0', file );
    }

/* write a row to a raw PPM file from pixelrow[], width cols */
#ifdef PBMPLUS_RAWBITS
static void
ppm_writeppmrowraw( file, pixelrow, cols, maxval )
    FILE* file;
    pixel* pixelrow;
    int cols;
    pixval maxval;
    {
    register int col;
    register pixel* pP;
    register pixval val;

    for ( col = 0, pP = pixelrow; col < cols; ++col, ++pP )
        {
        val = PPM_GETR( *pP );
#ifdef DEBUG
        if ( val > maxval )
            pm_error( "r value out of bounds (%u > %u)", val,
            maxval );
#endif /*DEBUG*/
        (void) putc( val, file );
        val = PPM_GETG( *pP );
#ifdef DEBUG
        if ( val > maxval )
            pm_error( "g value out of bounds (%u > %u)", val,
            maxval );
#endif /*DEBUG*/
```

```
            (void) putc( val, file );
            val = PPM_GETB( *pP );
#ifdef DEBUG
            if ( val > maxval )
                pm_error( "b value out of bounds (%u > %u)", val,
                maxval );
#endif /*DEBUG*/
            (void) putc( val, file );
            }
        }
#endif /*PBMPLUS_RAWBITS*/

/* write a row to a plain PPM file from pixelrow[], width cols */
static void
ppm_writeppmrowplain( file, pixelrow, cols, maxval )
    FILE* file;
    pixel* pixelrow;
    int cols;
    pixval maxval;
    {
    register int col, charcount;
    register pixel* pP;
    register pixval val;

    charcount = 0;
    for ( col = 0, pP = pixelrow; col < cols; ++col, ++pP )
        {
        if ( charcount >= 65 )
            {
            (void) putc( '\n', file );
            charcount = 0;
            }
        else if ( charcount > 0 )
            {
            (void) putc( ' ', file );
            (void) putc( ' ', file );
            charcount += 2;
            }
        val = PPM_GETR( *pP );
#ifdef DEBUG
        if ( val > maxval )
            pm_error( "r value out of bounds (%u > %u)", val,
            maxval );
#endif /*DEBUG*/
        putus( val, file );
        (void) putc( ' ', file );
        val = PPM_GETG( *pP );
#ifdef DEBUG
        if ( val > maxval )
```

```
                pm_error( "g value out of bounds (%u > %u)", val,
                maxval );
#endif /*DEBUG*/
            putus( val, file );
            (void) putc( ' ', file );
            val = PPM_GETB( *pP );
#ifdef DEBUG
            if ( val > maxval )
                pm_error( "b value out of bounds (%u > %u)", val,
                maxval );
#endif /*DEBUG*/
            putus( val, file );
            charcount += 11;
            }
        if ( charcount > 0 )
            (void) putc( '\n', file );
    }
```

Finally, ppm_writeppmrow( ) and ppm_writeppm( ) put all of the above routines together to write rows and entire files.

```
/*
 * write a row to a PPM file from pixelrow[], width cols, raw
 * unless maxval too big or forceplain set
 */
#if __STDC__
void
ppm_writeppmrow( FILE* file, pixel* pixelrow, int cols,
pixval maxval, int forceplain )
#else /*__STDC__*/
void
ppm_writeppmrow( file, pixelrow, cols, maxval, forceplain )
    FILE* file;
    pixel* pixelrow;
    int cols;
    pixval maxval;
    int forceplain;
#endif /*__STDC__*/
    {
#ifdef PBMPLUS_RAWBITS
    if ( maxval <= 255 && ! forceplain )
        ppm_writeppmrowraw( file, pixelrow, cols, maxval );
    else
        ppm_writeppmrowplain( file, pixelrow, cols, maxval );
#else /*PBMPLUS_RAWBITS*/
    ppm_writeppmrowplain( file, pixelrow, cols, maxval );
#endif /*PBMPLUS_RAWBITS*/
    }
```

```
/*
 * write PPM file from pixels[][], size cols by rows, max pixel val
 * maxval, plain unless maxval too big or forceplain set
 */
#if __STDC__
void
ppm_writeppm( FILE* file, pixel** pixels, int cols, int rows,
pixval maxval, int forceplain )
#else /*__STDC__*/
void
ppm_writeppm( file, pixels, cols, rows, maxval, forceplain )
    FILE* file;
    pixel** pixels;
    int cols, rows;
    pixval maxval;
    int forceplain;
#endif /*__STDC__*/
    {
    int row;

    ppm_writeppminit( file, cols, rows, maxval, forceplain );

    for ( row = 0; row < rows; ++row )
        ppm_writeppmrow( file, pixels[row], cols, maxval,
        forceplain );
    }
```

## 9.4 COLOR MAPS

Although PPM files are true color, with no color maps involved, most of the other color formats we handle do use color maps, so we define some routines here to handle them.

### Color Map Data Structures

There are two major data structures for handling color maps, color histograms, and color hash tables. Both are, essentially, maps from a set of colors to a set of small integers.

Color histograms (not a great name) are the simpler form. A colorhist_vector is an array (or pointer to one) of colorhist_item structures. Each item contains a color and a value, the value being an application-specific number that sometimes is the position in the color map and sometimes is used to hold a count of the number of pixels that use that color.

A colorhist_list is an item and a pointer to the next item that support linked lists of color items used in color histograms.

Entries in a colorhist_vector are indexed by item number, and entries in a colorlist_list are accessed sequentially by following the links.

Each colorhist_vector is allocated with a fixed number of entries, corresponding to the maximum number of colors allowed in the color map—typically 256.

```
/* ppmcmap.h - header file for colormap routines in libppm
*/

/* Color histogram stuff. */

typedef struct colorhist_item* colorhist_vector;
struct colorhist_item
    {
    pixel color;
    int value;
    };

typedef struct colorhist_list_item* colorhist_list;
struct colorhist_list_item
    {
    struct colorhist_item ch;
    colorhist_list next;
    };

colorhist_vector ppm_computecolorhist ARGS(( pixel** pixels,
int cols, int rows, int maxcolors, int* colorsP ));
/* Returns a colorhist *colorsP long (with space allocated for
** maxcolors). */

void ppm_addtocolorhist ARGS(( colorhist_vector chv, int* colorsP,
int maxcolors, pixel* colorP, int value, int position ));

void ppm_freecolorhist ARGS(( colorhist_vector chv ));
```

Finding a particular color in a colorhist_vector can be a slow process, since one has to search sequentially down the vector. Color hash tables provide relatively quick access by color. A colorhash_table is a vector of pointers colorhist_lists. Each color in the hash table has a position determined by hashing the components in the color. All of the colors in the table that hash to the same slot are linked into a list.

The routine that builds a color hash table takes as an argument a maximum number of entries to use even though, unlike colorhist_vectors, hash tables have no built-in size limit. The point of limiting the number of colors is that all of the applications have a maximum color-map size that they can handle, and if an image has more colors than that, there's no point in building the map since the application couldn't handle it anyway.

```
/* Color hash table stuff. */

typedef colorhist_list* colorhash_table;
```

```
colorhash_table ppm_computecolorhash ARGS(( pixel** pixels,
int cols, int rows, int maxcolors, int* colorsP ));

int
ppm_lookupcolor ARGS(( colorhash_table cht, pixel* colorP ));

colorhist_vector ppm_colorhashtocolorhist ARGS((
  colorhash_table cht, int maxcolors ));
colorhash_table ppm_colorhisttocolorhash ARGS((
  colorhist_vector chv, int colors ));

int ppm_addtocolorhash ARGS(( colorhash_table cht, pixel* colorP,
  int value ));
/* Returns -1 on failure. */

colorhash_table ppm_alloccolorhash ARGS(( void ));

void ppm_freecolorhash ARGS(( colorhash_table cht ));
```

## Color Map Building Routines

We start by defining some parameters for the color hash tables. The hash table size is 20,023 (a large prime number) for systems with plenty of memory and 263 (a considerably smaller prime number) for memory-limited MS-DOS systems. Colors are hashed into hash table entry numbers by the macro ppm_hashpixel( ), which combines the components into a single number, if they aren't already stored that way, and returns that value modulo to the hash size.

```
/* libppm3.c - ppm utility library part 3
*/

#include "ppm.h"
#include "ppmcmap.h"
#include "libppm.h"

#if MSDOS
#define HASH_SIZE 263        /* slower, but a lot smaller */
#else
#define HASH_SIZE 20023
#endif /* MSDOS */

#ifdef PPM_PACKCOLORS
#define ppm_hashpixel(p) ( ( (int) (p) & 0x7fffffff ) % HASH_SIZE )
#else /*PPM_PACKCOLORS*/
#define ppm_hashpixel(p) ( ( ( (long) PPM_GETR(p) * 33023 + (long)
PPM_GETG(p) * 30013 + (long) PPM_GETB(p) * 27011 ) & 0x7fffffff )
% HASH_SIZE )
#endif /*PPM_PACKCOLORS*/
```

To create a color map, one typically uses ppm_computecolorhash( ) to make a hash table of all the colors in the image, then turns that into a colorhist_vector, which is the color map. The routine ppm_computecolorhist( ) does those two steps and calls other routines, which we examine next.

```
/*
 * Make colorhist vector from array pixels[][], size cols by rows,
 * vector size maxcolors, store actual number of colors into
 * *colorsP and return vector as value
 */
colorhist_vector
ppm_computecolorhist( pixels, cols, rows, maxcolors, colorsP )
    pixel** pixels;
    int cols, rows, maxcolors;
    int* colorsP;
    {
    colorhash_table cht;
    colorhist_vector chv;

    cht = ppm_computecolorhash( pixels, cols, rows, maxcolors,
    colorsP );
    if ( cht == (colorhash_table) 0 )
        return (colorhist_vector) 0;
    chv = ppm_colorhashtocolorhist( cht, maxcolors );
    ppm_freecolorhash( cht );
    return chv;
    }
```

The routine ppm_addtocolorhist( ) puts an entry into a color histogram at a specified position, moving the entry to that position if it's already in the table or creating a new entry at that position if it isn't. This routine exists to support formats that require particular colors in particular positions in the table, most often black in position zero.

```
/*
 * Add color entry *colorP to color histogram chv, number of entries
 * *colorsP, value field value, position is position
 */
void
ppm_addtocolorhist( chv, colorsP, maxcolors, colorP, value,
position )
    colorhist_vector chv;
    pixel* colorP;
    int* colorsP;
    int maxcolors, value, position;
    {
    int i, j;
```

```
/* Search colorhist for the color. */
for ( i = 0; i < *colorsP; ++i )
    if ( PPM_EQUAL( chv[i].color, *colorP ) )
        {
        /* Found it - move to new slot. */
        if ( position > i )
            {
            for ( j = i; j < position; ++j )
                chv[j] = chv[j + 1];
            }
        else if ( position < i )
            {
            for ( j = i; j > position; --j )
                chv[j] = chv[j - 1];
            }
        chv[position].color = *colorP;
        chv[position].value = value;
        return;
        }
if ( *colorsP < maxcolors )
    {
    /* Didn't find it, but there's room to add it; so do so. */
    for ( i = *colorsP; i > position; --i )
        chv[i] = chv[i - 1];
    chv[position].color = *colorP;
    chv[position].value = value;
    ++(*colorsP);
    }
}
```

The best way to map the colors in an image is with ppm_computecolorhash( ). It makes an empty hash table, then sweeps through all of the pixels in an image, adding all of the colors in the image to the table. The value stored with each color is a count of the number of pixels of that color in the image. These counts can be useful for creating a reduced color map, discarding less used colors, or producing a "prioritized" color map. Some systems, notably Microsoft Windows, display color-mapped images using hardware that doesn't always have as many colors available as are in the map; in that case it starts at the beginning of the map and allocates as many colors as it can; so for best image display the most used colors should appear first.

```
/*
 * Make a hash table of all colors in array, pixels[][], size cols
 * by rows, maximum colors allowed maxcolors, store actual number
 * of colors in *colorsP, and return the hash table.
 */
colorhash_table
```

```
ppm_computecolorhash( pixels, cols, rows, maxcolors, colorsP )
    pixel** pixels;
    int cols, rows, maxcolors;
    int* colorsP;
    {
    colorhash_table cht;
    register pixel* pP;
    colorhist_list chl;
    int col, row, hash;

    cht = ppm_alloccolorhash( );
    *colorsP = 0;

    /* Go through the entire image, building a hash table
     * of colors. */
    for ( row = 0; row < rows; ++row )
        for ( col = 0, pP = pixels[row]; col < cols; ++col, ++pP )
            {
            hash = ppm_hashpixel( *pP );
            for ( chl = cht[hash]; chl != (colorhist_list) 0;
            chl = chl->next )
                if ( PPM_EQUAL( chl->ch.color, *pP ) )
                    break;
            if ( chl != (colorhist_list) 0 )
                ++(chl->ch.value);
            else
                {
                if ( ++(*colorsP) > maxcolors )
                    {
                    ppm_freecolorhash( cht );
                    return (colorhash_table) 0;
                    }
                chl = (colorhist_list) malloc(
                sizeof(struct colorhist_list_item) );
                if ( chl == 0 )
                    pm_error( "out of memory computing hash
                    table" );
                chl->ch.color = *pP;
                chl->ch.value = 1;
                chl->next = cht[hash];
                cht[hash] = chl;
                }
            }

return cht;
}
```

A few low-level routines are available to support both the higher-level color map routines and the application code that manipulates its own color maps. To create a new hash table, call ppm_alloccolorhash( ).

```
/* create and return a fresh, empty hash table */
colorhash_table
ppm_alloccolorhash( )
    {
    colorhash_table cht;
    int i;

    cht = (colorhash_table) malloc( HASH_SIZE *
    sizeof(colorhist_list) );
    if ( cht == 0 )
        pm_error( "out of memory allocating hash table" );

    for ( i = 0; i < HASH_SIZE; ++i )
        cht[i] = (colorhist_list) 0;

    return cht;
    }
```

To add a new entry to a hash table, call ppm_addtocolorhash( ). It doesn't check whether the color is already present, so either application code must already have checked whether the color is in the table, or (more commonly) it must be reading from a list of colors known to be unique.

```
/* add color in *colorP, which must not already be present,
 * to hash table cht, set value field to value
 */
int
ppm_addtocolorhash( cht, colorP, value )
    colorhash_table cht;
    pixel* colorP;
    int value;
    {
    register int hash;
    register colorhist_list chl;

    chl = (colorhist_list) malloc(
    sizeof(struct colorhist_list_item) );
    if ( chl == 0 )
        return -1;
    hash = ppm_hashpixel( *colorP );
    chl->ch.color = *colorP;
    chl->ch.value = value;
    chl->next = cht[hash];
    cht[hash] = chl;
    return 0;
    }
```

Once one has created a hash table, ppm_colorhashtocolorhist( ) turns it to a color histogram by enumerating all of the colors in the hash table. This implicitly assigns color-map entry numbers to colors that correspond to entry numbers in the histogram.

```
/* Turn color hash table cht to a color hist of length maxcolors */
colorhist_vector
ppm_colorhashtocolorhist( cht, maxcolors )
    colorhash_table cht;
    int maxcolors;
    {
    colorhist_vector chv;
    colorhist_list chl;
    int i, j;

    /* Now collate the hash table into a simple colorhist array. */
    chv = (colorhist_vector) malloc( maxcolors *
    sizeof(struct colorhist_item) );
    /* (Leave room for expansion by caller.) */
    if ( chv == (colorhist_vector) 0 )
        pm_error( "out of memory generating histogram" );

    /* Loop through the hash table. */
    j = 0;
    for ( i = 0; i < HASH_SIZE; ++i )
        for ( chl = cht[i]; chl != (colorhist_list) 0;
        chl = chl->next )
            {
            /* Add the new entry. */
            chv[j] = chl->ch;
            ++j;
            }

    /* All done. */
    return chv;
    }
```

The inverse function is ppm_colorhisttocolorhash( ), which makes an entry hash table and then inserts all of the colors in the colorhist_vector into the hash table. The value fields in the hash table contain the entry numbers in the vector, which are usually used as the color numbers in the color map.

This means that to create a color map for a particular image, one creates a color hash table, turns it into a colorhist_vector to assign entry numbers, then turns it back into a hash table to be able to find the numbers as one processes the colors in the image. This process could certainly be sped up, but since the color map is created only once per image, the performance gain wouldn't be great. The slowest part of the operation is generally the initial scan of the entire image, which wouldn't be affected at all.

```
/*
 * Turn color histogram chv with number of valid entries colors
 * into a hash table, storing the histogram entry numbers as the
 * value field in the hash table
 */
colorhash_table
ppm_colorhisttocolorhash( chv, colors )
    colorhist_vector chv;
    int colors;
    {
    colorhash_table cht;
    int i, hash;
    pixel color;
    colorhist_list chl;

    cht = ppm_alloccolorhash( );

    for ( i = 0; i < colors; ++i )
        {
        color = chv[i].color;
        hash = ppm_hashpixel( color );
        for ( chl = cht[hash]; chl != (colorhist_list) 0;
        chl = chl->next )
            if ( PPM_EQUAL( chl->ch.color, color ) )
                pm_error(
                    "same color found twice - %d %d %d",
                    PPM_GETR(color),
                    PPM_GETG(color), PPM_GETB(color) );
        chl = (colorhist_list) malloc(
        sizeof(struct colorhist_list_item) );
        if ( chl == (colorhist_list) 0 )
            pm_error( "out of memory" );
        chl->ch.color = color;
        chl->ch.value = i;
        chl->next = cht[hash];
        cht[hash] = chl;
        }

    return cht;
    }
```

Once one has the color hash table, ppm_lookupcolor( ) finds the value (colormap entry number) for any color.

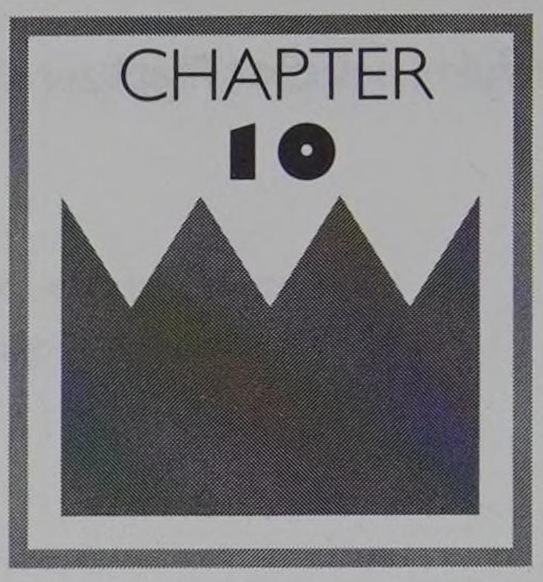
```
/*
 * lookup color *colorP in hash table cht and return corresponding
 * value
 */
```

```
int
ppm_lookupcolor( cht, colorP )
    colorhash_table cht;
    pixel* colorP;
    {
    int hash;
    colorhist_list chl;

    hash = ppm_hashpixel( *colorP );
    for ( chl = cht[hash]; chl != (colorhist_list) 0;
    chl = chl->next )
        if ( PPM_EQUAL( chl->ch.color, *colorP ) )
            return chl->ch.value;

    return -1;
    }
```

Finally, when one is done with color maps, colorhist_vectors can be freed with ppm_freecolorhist( ) and hash tables with ppm_freecolorhash( ).

```
/* free colorhist vector chv */
void
ppm_freecolorhist( chv )
    colorhist_vector chv;
    {
    free( (char*) chv );
    }

/* free color hash table cht and all entries */
void
ppm_freecolorhash( cht )
    colorhash_table cht;
    {
    int i;
    colorhist_list chl, chlnext;

    for ( i = 0; i < HASH_SIZE; ++i )
        for ( chl = cht[i]; chl != (colorhist_list) 0; chl =
              chlnext )
            {
            chlnext = chl->next;
            free( (char*) chl );
            }
    free( (char*) cht );
    }
```

CHAPTER 10



# Libraries for General Pixel Images

The last set of bitmap library routines are PNM (aNy Map), which support applications that support reading and writing any of the PBM, PGM, or PPM files. They work in much the same way that the PPM libraries do and use the same in-memory data format, although they rename the component type from pixval to xelval and the pixel type from pixel to xel. The difference from PPM is that, when reading a PBM or PGM file, they store the data unmodified in the first component field of the xel, so it can be manipulated in its native form. PNM also lets applications specify the format to use when writing a file, so they can write any of the PBM, PGM, or PPM files. The PNM routines are of use to two types of applications. The first type doesn't examine the pixel data, for example, an application that flips an image from left to right, which reverses the order of pixels in each row but which doesn't need to interpret the pixel values. The other type is an application that converts to or from formats, such as TIFF, which have multiple subformats. Depending on the subformat, a particular TIFF file might be most appropriately translated to a PBM, PGM, or PPM file. Conversely, a PBM file would be translated to one TIFF subformat, a PGM file to a second subformat, and a PPM file to a third. The application has to determine the appropriate format to use on the basis of the contents of the file.

## 10.1 PNM HEADERS

The PNM header file is quite short, because most of the formats are actually the same as PPM. It defines the component and pixel types xel and xelval. Two new macros, PNM_GET1(p) and PNM_ASSIGN1(p, v), read and write single component (i.e., PBM and PGM) values to and from xels. For PPM values, the existing PPM_GETx( ) and PPM_ASSIGN( ) suffice. The maximum value and format type macros are defined to be the same as for PPM.

There are no new magic numbers, since there is no PNM format per se.

```
/* pnm.h - header file for libpnm portable anymap library
*/

#ifndef _PNM_H_
#define _PNM_H_

#include "ppm.h"
typedef pixel xel;
typedef pixval xelval;
#define PNM_MAXMAXVAL PPM_MAXMAXVAL
#define PNM_GET1(x) PPM_GETB(x)
#define PNM_ASSIGN1(x,v) PPM_ASSIGN(x,0,0,v)
#define PNM_EQUAL(x,y) PPM_EQUAL(x,y)
#define PNM_FORMAT_TYPE(f) PPM_FORMAT_TYPE(f)
```

We define a few conventional type names used in programs from BSD Unix. Several PNM programs, notably the TIFF conversion routines, were originally written for BSD and use type names such as u_char for unsigned char and u_long for unsigned long.

```
#ifndef BSD
typedef         unsigned char u_char;
typedef         unsigned short u_short;
typedef         unsigned int u_int;
typedef         unsigned long u_long;
#endif /* BSD */
```

Finally, we declare the public PNM routines. The storage allocation routines are, as always, macros that call the corresponding pm_ routines.

```
/* Declarations of routines. */

void pnm_init ARGS(( int* argcP, char* argv[] ));

#define pnm_allocarray( cols, rows ) \
        ((xel**) pm_allocarray( cols, rows, sizeof(xel) ))
#define pnm_allocrow( cols ) ((xel*) \
        pm_allocrow( cols, sizeof(xel) ))
#define pnm_freearray( xels, rows ) \
        pm_freearray( (char**) xels, rows )
#define pnm_freerow( xelrow ) pm_freerow( (char*) xelrow )

xel** pnm_readpnm ARGS(( FILE* file, int* colsP, int* rowsP,
xelval* maxvalP, int* formatP ));
void pnm_readpnminit ARGS(( FILE* file, int* colsP, int* rowsP,
xelval* maxvalP, int* formatP ));
void pnm_readpnmrow ARGS(( FILE* file, xel* xelrow, int cols,
xelval maxval, int format ));
```

```
void pnm_writepnm ARGS(( FILE* file, xel** xels, int cols,
int rows, xelval maxval, int format, int forceplain ));
void pnm_writepnminit ARGS(( FILE* file, int cols, int rows,
xelval maxval, int format, int forceplain ));
void pnm_writepnmrow ARGS(( FILE* file, xel* xelrow, int cols,
xelval maxval, int format, int forceplain ));

extern xelval pnm_pbmmaxval;
/* This is the maxval used when a PNM program reads a PBM file.
** Normally it is 1; however, for some programs, a larger value
** gives better results
*/

#endif /*_PNM_H_*/
```

## 10.2 PNM INITIALIZATION

The PNM initialization routine, not surprisingly, calls just the PPM version.

```
/* libpnm1.c - pnm utility library part 1
*/

#include "pnm.h"

#include "ppm.h"
#include "libppm.h"

#include "pgm.h"
#include "libpgm.h"

#include "pbm.h"
#include "libpbm.h"

/* pnm-specific initialization, same arguments as pm_init */
void
pnm_init( argcP, argv )
    int* argcP;
    char* argv[];
    {
    ppm_init( argcP, argv );
    }
```

## 10.3 READING PNM FILES

The PNM reading code is practically identical to the PPM reading code. An application can either call pnm_readpnm( ) to read an entire file at once or else call pnm_readpnminit( ) followed by a call to pnm_readpnmrow( ) for each row in the image.

The primary difference is that PBM and PGM are stored in native form in the first component of each xel rather than converted to PPM.

```
xelval pnm_pbmmaxval = 1;

/*
 * prepare to read PPM, PGM or PBM file, store size in *colsP and
 * *rowsP, store max pixel value in *maxvalP, store actual format
 * in *formatP
 */
void
pnm_readpnminit( file, colsP, rowsP, maxvalP, formatP )
    FILE* file;
    int* colsP;
    int* rowsP;
    int* formatP;
    xelval* maxvalP;
    {
    gray gmaxval;

    /* Check magic number. */
    *formatP = pbm_readmagicnumber( file );
    switch ( PNM_FORMAT_TYPE(*formatP) )
        {
        case PPM_TYPE:
        ppm_readppminitrest( file, colsP, rowsP, (pixval*)
        maxvalP );
        break;

        case PGM_TYPE:
        pgm_readpgminitrest( file, colsP, rowsP, &gmaxval );
        *maxvalP = (xelval) gmaxval;
        break;

        case PBM_TYPE:
        pbm_readpbminitrest( file, colsP, rowsP );
        *maxvalP = pnm_pbmmaxval;
        break;

        default:
        pm_error( "bad magic number - not a ppm, pgm, or
        pbm file" );
        }
    }
```

As each row is read, ppm_readppmrow( ) takes appropriate action depending on the file format being read. For plain PPM files, it reads triplets of integers representing the red, green, and blue components using pbm_getint( ), defined

in Chapter 7. For raw PPM files, it reads binary bytes with pbm_getrawbyte( ), which we saw in Chapter 9.

For files in PGM and PBM formats, it allocates storage for one row in the file's native format, calls the native row reading routine, stores the values into the xel row, and frees the row. A useful optimization would be to hold on to the storage for the PBM or PGM row, since this routine is usually called in a tight loop.

```
/*
 * Read a row of pixels from file into xelrow[], width cols,
 * max component value maxval, file format format
 */
#if __STDC__
void
pnm_readpnmrow( FILE* file, xel* xelrow, int cols, xelval maxval,
int format )
#else /*__STDC__*/
void
pnm_readpnmrow( file, xelrow, cols, maxval, format )
    FILE* file;
    xel* xelrow;
    xelval maxval;
    int cols, format;
#endif /*__STDC__*/
    {
    register int col;
    register xel* xP;
    gray* grayrow;
    register gray* gP  ;
    bit* bitrow;
    register bit* bP;

    switch ( PNM_FORMAT_TYPE(format) )
        {
case PPM_TYPE:
ppm_readppmrow( file, (pixel*) xelrow, cols, (pixval)
maxval,
format );
break;

case PGM_TYPE:
grayrow = pgm_allocrow( cols );
pgm_readpgmrow( file, grayrow, cols, (gray) maxval,
format );
for ( col = 0, xP = xelrow, gP = grayrow; col < cols;
++col, ++xP, ++gP )
    PNM_ASSIGN1( *xP, *gP );
pgm_freerow( grayrow );
break;
```

```
        case PBM_TYPE:
        bitrow = pbm_allocrow( cols );
        pbm_readpbmrow( file, bitrow, cols, format );
        for ( col = 0, xP = xelrow, bP = bitrow; col < cols;
        ++col, ++xP, ++bP )
            PNM_ASSIGN1( *xP, *bP == PBM_BLACK ? 0:
            pnm_pbmmaxval );
        pbm_freerow( bitrow );
        break;

        default:
        pm_error( "can't happen" );
        }
    }
```

Finally, we see pnm_readpnm( ), which puts all the above routines together.

```
/*
 * Read a whole array from file, store size into *colsP and *rowsP,
 *  max component value into *maxvalP, return pointer to array.
 */
xel**
pnm_readpnm( file, colsP, rowsP, maxvalP, formatP )
    FILE* file;
    int* colsP;
    int* rowsP;
    int* formatP;
    xelval* maxvalP;
    {
    xel** xels;
    int row;

    pnm_readpnminit( file, colsP, rowsP, maxvalP, formatP );

    xels = pnm_allocarray( *colsP, *rowsP );

    for ( row = 0; row < *rowsP; ++row )
        pnm_readpnmrow( file, xels[row], *colsP, *maxvalP,
        *formatP );

    return xels;
    }
```

## 10.4 WRITING PNM FILES

The PNM writing routines are somewhat different from any of the previous writing routines since they can write any of the six formats.

The routines are still largely parallel to those in the other libraries. An application can either write an entire file by calling pnm_writepnm( ) or write the file row-by-row by calling pnm_writepnminit( ) followed by calling pnm_writepnmrow( ) for each row. Each takes an argument identifying the file type to write.

The argument "forceplain" is interpreted as it is by the previous routines and, again, is rarely used.

```
/* libpnm2.c - pnm utility library part 2
*/

#include "pnm.h"

#include "ppm.h"
#include "libppm.h"

#include "pgm.h"
#include "libpgm.h"

#include "pbm.h"
#include "libpbm.h"

/*
 * Start writing file, size cols x rows, max component value maxval,
 * given format, raw unless maxval too big or forceplain set
 */
#if __STDC__
void
pnm_writepnminit( FILE* file, int cols, int rows, xelval maxval,
int format, int forceplain )
#else /*__STDC__*/
void
pnm_writepnminit( file, cols, rows, maxval, format, forceplain )
    FILE* file;
    int cols, rows, format;
    xelval maxval;
    int forceplain;
#endif /*__STDC__*/
    {
    switch ( PNM_FORMAT_TYPE(format) )
        {
        case PPM_TYPE:
        ppm_writeppminit( file, cols, rows, (pixval) maxval,
        forceplain );
        break;

        case PGM_TYPE:
        pgm_writepgminit( file, cols, rows, (gray) maxval,
        forceplain );
        break;
```

```
        case PBM_TYPE:
        pbm_writepbminit( file, cols, rows, forceplain );
        break;

        default:
        pm_error( "can't happen" );
        }
    }
```

Writing the rows primarily involves switching off to the appropriate native writing routines. The PPM writing routines can be called directly, since the data format for an xel is the same as for a PPM pixel. For PBM and PGM the data first must be copied to a row in the native format, then that row is written, and finally freed.

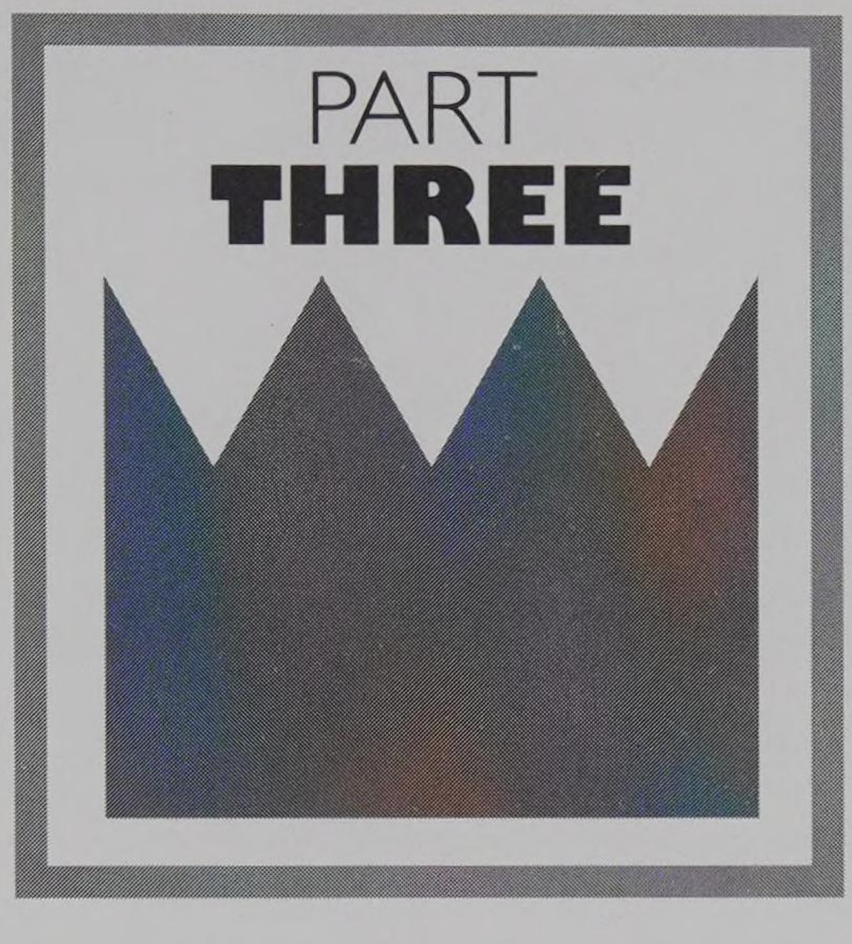
```
/*
 * write a row to a file from xelrow[], width cols, given format,
 * raw unless maxval too big or forceplain set
 */
#if __STDC__
void
pnm_writepnmrow( FILE* file, xel* xelrow, int cols, xelval maxval,
int format, int forceplain )
#else /*__STDC__*/
void
pnm_writepnmrow( file, xelrow, cols, maxval, format, forceplain )
    FILE* file;
    xel* xelrow;
    int cols, format;
    xelval maxval;
    int forceplain;
#endif /*__STDC__*/
{
register int col;
register xel* xP;
gray* grayrow;
register gray* gP;
bit* bitrow;
register bit* bP;

switch ( PNM_FORMAT_TYPE(format) )
    {
    case PPM_TYPE:
    ppm_writeppmrow( file, (pixel*) xelrow, cols, (pixval)
    maxval,
    forceplain );
    break;
```

```
        case PGM_TYPE:
        grayrow = pgm_allocrow( cols );
        for ( col = 0, gP = grayrow, xP = xelrow; col < cols;
        ++col, ++gP, ++xP )
            *gP = PNM_GET1( *xP );
        pgm_writepgmrow( file, grayrow, cols, (gray) maxval,
        forceplain );
        pgm_freerow( grayrow );
        break;

        case PBM_TYPE:
        bitrow = pbm_allocrow( cols );
        for ( col = 0, bP = bitrow, xP = xelrow;
        col < cols; ++col, ++bP, ++xP )
            *bP = PNM_GET1( *xP ) == 0 ? PBM_BLACK : PBM_WHITE;
        pbm_writepbmrow( file, bitrow, cols, forceplain );
        pbm_freerow( bitrow );
        break;

        default:
        pm_error( "can't happen" );
        }
    }
```

Finally, pnm_writepnm( ) puts all the above routines together to write entire files.

```
/*
 * write file from xels[][], size cols by rows, max pixel
 * val maxval, given format, plain unless maxval too big
 * or forceplain set
 */
#if __STDC__
void
pnm_writepnm( FILE* file, xel** xels, int cols, int rows,
xelval maxval, int format, int forceplain )
#else /*__STDC__*/
void
pnm_writepnm( file, xels, cols, rows, maxval, format,
              forceplain )
    FILE* file;
    xel** xels;
    xelval maxval;
    int cols, rows, format;
    int forceplain;
#endif /*__STDC__*/
    {
    int row;
```

```
pnm_writepnminit( file, cols, rows, maxval, format,
forceplain );

for ( row = 0; row < rows; ++row )
    pnm_writepnmrow( file, xels[row], cols, maxval,
    format,
    forceplain );
}
```




# Run-Length Compressed Formats

CHAPTER 11

# MacPaint

MacPaint file format was originally defined to store images in the MacPaint program distributed with the Apple Macintosh. Because the format is so simple, and because every Mac has a copy of MacPaint, it has become an extremely common format for images that it can handle.

## 11.1 OVERVIEW

MacPaint files are extremely simple. Each image is 576 pixels wide and 720 pixels high, corresponding to a full page of data at 72 dots per inch, the resolution of the screen and printer available at the time. Each pixel is a single bit, 0 for white and 1 for black. The pixels for each row are stored left to right, row by row. Each row is compressed using PackBits compression (described in Chapter 4.)

The file has a 512-byte header containing information of use to the MacPaint program but not necessary to interpret the image. Other programs usually ignore the header and write it as zeros. Some file transfer programs add an extra 128-byte header to the file, which also needs to be ignored.

The header file for the MacPaint handling code is very simple, reflecting the simplicity of the file format.

```
/* macp.h - header file for MacPaint files */

#ifndef _MACP_H_
#define _MACP_H_

#define         HEADER_LENGTH         512
#define         MAX_LINES          720
#define         BYTES_WIDE          72
#define MAX_COLS          576          /* = BYTES_WIDE * 8 */

#endif /*_MACP_H_*/
```

## 11.2 READING MACPAINT FILES

Reading a MacPaint file involves three steps:

- Skip the header.
- Read compressed chunks.
- Expand the compressed chunks and store the pixels.

The routine ReadMacPaintFile( ) actually reads the file. It takes four arguments:

- A FILE pointer for the file to read.
- The number of extra bytes of header to skip, normally zero.
- The place to store the number of rows actually read.
- An output array into which to store the image.

ReadMacPaintFile( ) first reads and ignores the appropriate number of bytes for the header. Then it enters a loop, where it reads a PackBits length byte and either reads and stores a group of bytes from the file into the output array, or it reads 1 byte and stores it repeatedly into the output. The routine exits when it has read all 720 rows of the image.

This creates an array in memory containing a packed image with 8 pixels per byte.

```
/* macptopbm.c */
static void
ReadMacPaintFile( file, extraskip, scanLineP, Pic )
    FILE* file;
    int extraskip;
    int* scanLineP;
unsigned char Pic[MAX_LINES][BYTES_WIDE];
{
unsigned int i, j, k;
unsigned char ch;

/* Skip over the header. */
for ( i = 0; i < extraskip; i++ )
    getc( file );
for ( i = 0; i < HEADER_LENGTH; i++ )
    getc( file );

*scanLineP = 0;
k = 0;

while ( *scanLineP < MAX_LINES )
    {
```

```
ch = (unsigned char) getc( file );          /* Count byte */
i = (unsigned int) ch;
if ( ch < 0x80 )
    {          /* Unpack next (I+1) chars as is */
    for ( j = 0; j <= i; j++ )
        if ( *scanLineP < MAX_LINES )
            {
            Pic[*scanLineP][k++] = (unsigned char)
            getc( file );
            if ( ! (k %= BYTES_WIDE) )
                *scanLineP += 1;
            }
    }
else
    {          /* Repeat next char (2's comp I) times */
    ch = getc( file );
    for ( j = 0; j <= 256 - i; j++ )
        if ( *scanLineP < MAX_LINES )
            {
            Pic[*scanLineP][k++] = (unsigned char) ch;
            if ( ! (k %= BYTES_WIDE) )
                *scanLineP += 1;
            }
    }
}
```

Now that we have the data for a row, we unpack the bits into PBM format and write them out using the standard library routine.

```
bitrow = pbm_allocrow( cols );

for ( row = 0; row < rows; row++ )
{
for ( bcol = 0; bcol < BYTES_WIDE; bcol++ )
    for ( i = 0; i < 8; i++ )
        bitrow[bcol * 8 + i] =
            ( (Pic[row][bcol] >> (7 - i)) & 1 ) ?
                PBM_BLACK : PBM_WHITE;
pbm_writepbmrow( stdout, bitrow, cols, 0 );
}
```

## 11.3 WRITING MACPAINT FILES

Writing MacPaint files is slightly more complex than reading them, because the writer has to search for repeated byte sequences to encode using PackBits. The main routine to write a MacPaint file takes one argument, the array containing the uncompressed image to write.

The routine calls header( ) to write an empty header file, then for each row calls packit( ) to compress the row and then sendbytes( ) to write the compressed data to the file.

```
/* pbmtomacp.c */
static void writemacp ARGS(( bit **bits ));
static int packit ARGS(( bit *pb, bit *bits ));
static void filltemp ARGS(( bit *dest, bit *src ));
static void sendbytes ARGS(( bit *pb, register int npb ));
static void header ARGS(( void ));

static void
writemacp( bits )
bit **bits;
{ register int i;
  bit pb[MAX_COLS * 2];
  int npb;

  header();
  for( i=0; i < MAX_LINES; i++ )
  { npb = packit( pb, bits[i] );
    sendbytes( pb, npb );
  }
} /* writemacp */
```

Now we look at the routines that perform PackBits compression. Routine packit( ) takes two arguments, the place to store the packed string and the input row of uncompressed data. It calls filltemp( ) to convert a row of 1 byte per pixel data into 1 bit per pixel (i.e., 8 pixels per byte), then scans the bytes just created to find sequences of more than two repeating bytes. When it finds repeated sequences, it outputs a PackBits repeat chunk, otherwise it outputs an unrepeated chunk.

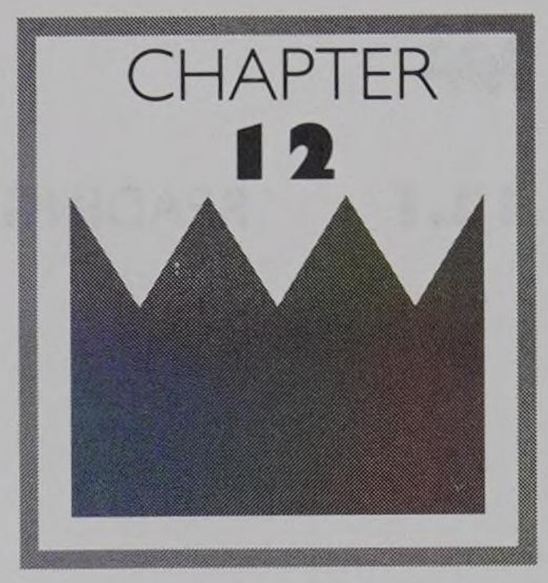
```
/* pack a line of bytes using PackBits
 * return value = # of bytes in pb
 */
static int
packit( pb, bits )
bit *pb, *bits;
{ register int charcount, npb, newcount, flg;
  bit temp[72];
  bit *count, *srcb, *destb, save;

  srcb = bits; destb = temp;
  filltemp( destb, srcb );
  srcb = temp;
  destb = pb;
  npb = 0;
```

```
    charcount = BYTES_WIDE;
    flg = EQUAL;
    while( charcount )
    { save = *srcb++;
      charcount--;
      newcount = 1;
      while( (*srcb == save) && charcount )
      { srcb++;
        newcount++;
        charcount--;
      }
      if( newcount > 2 )
      { count = destb++;
        *count = 257 - newcount;
        *destb++ = save;
        npb += 2;
        flg = EQUAL;
      }
      else
      { if( flg == EQUAL )
        { count = destb++;
          *count = newcount - 1;
          npb++;
        }
        else
          *count += newcount;
        while( newcount-- )
        { *destb++ = save;
          npb++;
        }
        flg = UNEQUAL;
    } }
    return npb;
} /* packit */

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

static void
filltemp( dest, src )
bit *dest, *src;
{ register unsigned char ch, zero, acht;
  register int i, j;

  zero = '\0';
  acht = 8;
  i = BYTES_WIDE;
  while( i-- )
  { ch = zero;
    j = acht;
```

```
    while( j-- )
    { ch <<= 1;
      if( *src++ )
        ch++;
    }
    *dest++ = ch;
  }
} /* filltemp */
```

There are two other helper routines, sendbytes( ), which copies its input to the file (and could better be written using fwrite( )), and header( ), which writes out the empty 512-byte header.

```
static void
sendbytes( pb, npb )
bit *pb;
register int npb;
{ register bit *b;

  b = pb;
  while( npb-- )
    (void) putc( *b++, fdout );
} /* sendbytes */

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

static void
header()
{ register int i;
  register char ch;

  /* header contains nothing ... */
  ch = '\0';
  for(i = 0; i < HEADER_LENGTH; i++ )
    (void) putc( ch, fdout );
} /* header */
```




# PCX

PCX files were originally designed by Zsoft Corporation to store images from their PC Paint program. Since PC Paint was one of the earliest paint programs for the PC, PCX format has become one of the most widely used. The format is a relatively straightforward bitmap image format with optional color maps and with some coding features that reflect peculiarities of some of the IBM screen adapters on which images were commonly displayed.

Each file starts with a 128-byte header describing the file, followed by lightly compressed image data. The header includes the number of image planes and the number of bits per pixel in each plane. For images with 16 colors or less, the header contains a color map. For those with more colors, the color map follows the image data. Although there are in principle dozens of possible combinations of numbers of planes and bits per pixel, in practice only a handful occur:

**1 bit, 1 plane** A single bit per pixel, like PBM.

**2, 3, or 4 planes, 1 bit** Color-mapped images with 4, 8, or 16 colors, stored by plane.

**1 bit, 2 to 8 planes** Color-mapped images with between 4 and 256 colors, stored by pixel

**8 bits, 3 planes** 24-bit true-color images. (The code here doesn't handle this relatively uncommon subformat.)

The most difficult aspect of decoding PCX files is determining which of the many subformats the image uses, since various versions of PC Paint have not been completely consistent in their use of header fields. Fortunately, the header fields that identify the bit and plane counts are invariably filled in correctly.

## 12.1 READING PCX FILES

Reading a PCX file involves four steps:

- Read the header.
- Read and decompress the data section.
- Read the color map (if it's at the end).
- Unscramble the data and apply the color map.

Our PCX reader starts with the usual assortment of constant definitions and forward function definitions:

```
/*
 * pcxtoppm.c - Converts from a PC Paintbrush PCX file to a
 * PPM file.
 */
#include        <stdio.h>
#include        "ppm.h"

#define PCX_MAGIC       0x0a  /* PCX magic number      */
#define PCX_HDR_SIZE    128   /* size of PCX header    */
#define PCX_256_COLORS  0x0c
                        /* magic number for 256 colors */

#define MAXCOLORS       256
#define MAXPLANES       4
#define PCX_MAXVAL      255

static void read_pcx_image ARGS(( FILE *fp, unsigned char *buf,
int BytesPerLine, int Planes, int Height ));
static void pcx_planes_to_pixels ARGS(( unsigned char *pixels,
unsigned char *bitplanes, int bytesperline, int planes,
int bitsperpixel ));
static void pcx_unpack_pixels ARGS(( unsigned char *pixels,
unsigned char *bitplanes, int bytesperline, int planes,
int bitsperpixel ));
static int GetByte ARGS(( FILE *fp ));
static int GetWord ARGS(( FILE *fp ));
```

The main( ) routine starts with a typical PPM initialization and a somewhat overimplemented way to determine whether the user passed a file name on the command line.

```
void
main(argc, argv)
    int         argc;
    char        *argv[];
```

```
{
    register int        i;
    FILE                *ifp;
    char                *ifname;
    int                 Version;
    int                 Xmin, Ymin, Xmax, Ymax;
    int                 Width, Height;
    register int        x, y;
    int                 Planes;
    int                 BitsPerPixel;
    int                 BytesPerLine;
    unsigned char       Red[MAXCOLORS], Green[MAXCOLORS],
                        Blue[MAXCOLORS];
    unsigned char       *pcximage;
    unsigned char       *pcxplanes;
    unsigned char       *pcxpixels;
    pixel               **pixels;

    ppm_init( &argc, argv );

    switch (argc)
    {
         case 1:
             ifname     = "standard input";
             ifp        = stdin;
             break;
         case 2:
             ifname     = argv[1];
             ifp        = pm_openr(ifname);
             break;
         default:
             pm_usage("[pcxfile]");
             break;
    }
```

Now we read the header from the PCX file, using the routine GetByte( ), which is defined later but is equivalent to the library routine pbm_getrawbyte( ). The header starts with a magic number of hex 0A, an encoding version number that must be 1 (they only defined one encoding scheme) and the size of the image.

```
    /*
     * read the PCX header
     */
    if (GetByte(ifp) != PCX_MAGIC)
         pm_error("%s is not a PCX file", ifname );

    Version     = GetByte(ifp);   /*  get version #  */

    if (GetByte(ifp) != 1)  /* check for PCX run length
```

```
                                                        encoding */
        pm_error("%s has unknown encoding scheme", ifname );

    BitsPerPixel= GetByte(ifp);
    Xmin        = GetWord(ifp);
    Ymin        = GetWord(ifp);
    Xmax        = GetWord(ifp);
    Ymax        = GetWord(ifp);

    Width       = (Xmax - Xmin) + 1;
    Height      = (Ymax - Ymin) + 1;

    (void) GetWord(ifp);    /* ignore horizontal resolution */
    (void) GetWord(ifp);    /* ignore vertical resolution */
```

The next thing in the header is the color map. If the image uses 16 colors or less, this is the actual color map. If it uses more, this map is a dummy and the real map is at the end of the file. We store the color map into the three vectors Red[ ], Green[ ], and Blue[ ] for later use.

```
    /*
     * get the 16-color color map
     */
    for (i = 0; i < 16; i++)
    {
        Red[i]    = GetByte(ifp);
        Green[i]  = GetByte(ifp);
        Blue[i]   = GetByte(ifp);
    }
```

Now we read the the number of planes which, along with the bits per pixel, determine the image format and validate that the format is indeed one that we can handle. We also read BytesPerLine, the number of bytes needed to store each row of pixels in the uncompressed image. (Each row is padded to a byte boundary if need be.)

```
    (void) GetByte(ifp);          /* skip reserved byte   */
    Planes      = GetByte(ifp);   /* # of color planes    */
    BytesPerLine= GetWord(ifp);   /* # of bytes per line  */
    (void) GetWord(ifp);          /* ignore palette info  */

    /*
     * check that we can handle this image format
     */
    switch (BitsPerPixel)
    {
        case 1:
        if (Planes > 4)
```

```
                pm_error("can't handle image with more than
                                                    4 planes");
            break;

        case 2:
        case 4:
        case 8:
            if (Planes == 1)
                break;
        default:
            pm_error("can't handle %d bits per pixel image with
                                                    %d planes",
                    BitsPerPixel,Planes);
    }
```

That's all the header information we need, so we seek to the end of the data and read and uncompress the stored image using read_pcx_image( ), shown later in this chapter.

```
/*
 * read the pcx format image
 */
fseek(ifp, (long)PCX_HDR_SIZE, 0);
pximage = (unsigned char *)pm_allocrow(BytesPerLine * Planes,
Height);
read_pcx_image(ifp, pximage, BytesPerLine, Planes, Height);
```

If the image uses more than 16 colors, the color map appears at the end of the file, following a signature byte of hex 0C. We store the color map into the three vectors Red[ ], Green[ ], and Blue[ ] for later use.

```
/*
 * 256 color images have their color map at the end of the
 * file preceded by a magic byte
 */
if (BitsPerPixel == 8)
{
    if (GetByte(ifp) != PCX_256_COLORS)
        pm_error("bad color map signature" );

    for (i = 0; i < MAXCOLORS; i++)
    {
        Red[i]      = GetByte(ifp);
        Green[i]    = GetByte(ifp);
        Blue[i]     = GetByte(ifp);
    }
}
```

Armed with the uncompressed image data and the color map, we can reconstruct the actual image, one row at a time. The pixel data goes into the PPM array pixels[ ][ ], and we use pcxpixels[ ] temporarily to store each row of unscrambled but still color-mapped pixels.

```
pixels      = ppm_allocarray(Width, Height);
pcxpixels   = (unsigned char *)pm_allocrow(Width+7, 1);

/*
 * convert the image
 */
for (y = 0; y < Height; y++)
{
     pcxplanes = pcximage + (y * BytesPerLine * Planes);
```

If the image is stored in a single plane, the image format is packed pixels, with 1, 2, 4, or 8 pixels per byte, depending on pixel size. In this case, pcx_unpack_pixels( ) unpacks each row of pixels.

If the image is stored in multiple planes, each pixel must be reconstructed from individual bits stored in each plane, using pcx_planes_to_pixels( ).

```
     if (Planes == 1)
     {
          pcx_unpack_pixels(pcxpixels, pcxplanes,
               BytesPerLine, Planes, BitsPerPixel);
     }
     else
     {
          pcx_planes_to_pixels(pcxpixels, pcxplanes,
               BytesPerLine, Planes, BitsPerPixel);
     }
```

Once we have reconstructed the pixel data, we look up the actual component data in the color map and store the actual image data.

```
          for (x = 0; x < Width; x++)
          {
               i = pcxpixels[x];
               PPM_ASSIGN(pixels[y][x], Red[i], Green[i], Blue[i]);
          }
     }
```

Having reconstructed the whole image, we write out a data to a PPM file and exit.

```
pm_close(ifp);

ppm_writeppm(stdout, pixels, Width, Height, (pixval) 255, 0 );
```

```
    pm_close(stdout);

    exit(0);
} /* main */
```

## Uncompressing the Stored Image Data

PCX files use a simple-minded compression scheme. A databyte with the high 2 bits set indicates a repeated byte, with the low 6 bits of that byte being the repeat count and the following byte the one to repeat. (This means that data bytes in range C0 to FF have to be stored as 1-byte repeating groups even if they aren't repeated.) Any other byte value represents itself.

```
/*
 * Read and unpack a file of PCX data from file fp, store in buf[],
 * with the image Height rows, with each row having Planes image
 * planes each BytesPerLine bytes
 */
static void
read_pcx_image(fp, buf, BytesPerLine, Planes, Height)
    FILE        *fp;
    unsigned char   *buf;
    int         BytesPerLine;
    int         Planes;
    int         Height;
{
    int         c;
    int         nbytes;
    int         count;

    nbytes      = BytesPerLine * Planes * Height;

    while (nbytes > 0)
    {
        c    = GetByte(fp);
        if ((c & 0xc0) != 0xc0)     /* repeated group * */
        {
            *buf++    = c;          /* no, just store it */
            --nbytes;
            continue;
        }

        count    = c & 0x3f;        /* extract count */
        c    = GetByte(fp);         /* get byte to repeat */
        if (count > nbytes)
            pm_error("repeat count spans end of image, count = %d,
                        nbytes = %d", count, nbytes);
```

```
            nbytes    -= count;    /* store it count times */
            while (--count >= 0)
                *buf++ = c;
        }
} /* read_pcx_image */
```

## Converting Data Planes to Pixels

If the image data is stored as image planes, the unpacked file data first contains all the low-order bits of all the pixels in a row, then all the next-to-low-order bits, and so forth, up to the number of planes, which is the same as the number of bits per pixel.

We unpack the pixels by first zeroing out the pixel buffer, then processing a plane at a time, distributing the bytes from that plane into the appropriate bit position in the pixels.

This code doesn't handle the case where there are multiple planes and multiple bits per pixel; fortunately, such PCX images are extremely uncommon.

```
/*
 * convert multi-plane format into 1 pixel per byte
 * from unpacked file data bitplanes[] into pixel row pixels[],
 * image Height rows, with each row having planes image planes each
 * bytesperline bytes
 */
static void
pcx_planes_to_pixels(pixels, bitplanes, bytesperline, planes,
bitsperpixel)
unsigned char   *pixels;
unsigned char   *bitplanes;
int             bytesperline;
int             planes;
int             bitsperpixel;
{
    int  i, j;
    int  npixels;
    unsigned char    *p;

    if (planes > 4)
         pm_error("can't handle more than 4 planes" );
    if (bitsperpixel != 1)
         pm_error("can't handle more than 1 bit per pixel" );
```

Here we clear the pixel buffer in preparation for OR-ing the bits in as they are extracted from planes.

```
    /*
     * clear the pixel buffer
```

```
     */
    npixels = (bytesperline * 8) / bitsperpixel;
    p    = pixels;
    while (--npixels >= 0)
        *p++ = 0;
```

Now iterate over the planes and distribute the bit-plane data into the pixels by OR-ing it into the pixel buffer.

```
    /*
     * do the format conversion
     */
    for (i = 0; i < planes; i++)
    {
        int pixbit, bits, mask;
        p    = pixels;
        pixbit    = (1 << i);    /* pixel bit for this plane */
        for (j = 0; j < bytesperline; j++)
        {
            bits = *bitplanes++; /* get 8 bits */
                                 /* ... and OR them into the pixel */
                                 /* buffer */
            for (mask = 0x80; mask != 0; mask >>= 1, p++)
                if (bits & mask)
                    *p |= pixbit;
        }
    }
} /* pcx_planes_to_pixels */
```

## Unpacking Packed Pixels

The single-plane, multiple-bit format is easier to handle. We need only unpack the pixels in each byte.

```
/*
 * convert packed pixel format into 1 pixel per byte
 * from unpacked file data bitplanes[] into pixel row pixels[],
 * image Height rows, with each row having planes image planes each
 * bytesperline bytes
 */
static void
pcx_unpack_pixels(pixels, bitplanes, bytesperline, planes,
bitsperpixel)
unsigned char   *pixels;
unsigned char   *bitplanes;
int             bytesperline;
int             planes;
int             bitsperpixel;
```

```
{
    register int        bits;

    if (planes != 1)
         pm_error("can't handle packed pixels with more than 1
                   plane" );

    if (bitsperpixel == 8)
                       /* 8 bits/pixel, no unpacking needed */
    {
        while (--bytesperline >= 0)
            *pixels++ = *bitplanes++;
    }
    else if (bitsperpixel == 4)
                      /* 4 bits/pixel, two pixels per byte */
    {
        while (--bytesperline >= 0)
        {
            bits        = *bitplanes++;
            *pixels++   = (bits >> 4) & 0x0f;
            *pixels++   = (bits     ) & 0x0f;
        }
    }
    else if (bitsperpixel == 2)
                     /* 2 bits/pixel, four pixels per byte */
    {
        while (--bytesperline >= 0)
        {
            bits        = *bitplanes++;
            *pixels++   = (bits >> 6) & 0x03;
            *pixels++   = (bits >> 4) & 0x03;
            *pixels++   = (bits >> 2) & 0x03;
            *pixels++   = (bits     ) & 0x03;
        }
    }
    else if (bitsperpixel == 1)
                    /* 1 bit/pixel, 8 pixels per byte */
    {
        while (--bytesperline >= 0)
        {
            bits        = *bitplanes++;
                    /* 1/0 for true/false easier than shifting */
            *pixels++   = ((bits & 0x80) != 0);
            *pixels++   = ((bits & 0x40) != 0);
            *pixels++   = ((bits & 0x20) != 0);
            *pixels++   = ((bits & 0x10) != 0);
            *pixels++   = ((bits & 0x08) != 0);
            *pixels++   = ((bits & 0x04) != 0);
```

```
                *pixels++   = ((bits & 0x02) != 0);
                *pixels++   = ((bits & 0x01) != 0);
            }
        }
} /* pcx_unpack_pixels */
```

Finally, we have the two small auxiliary routines, GetByte( ) and GetWord( ), similar to the PBM pbm_getrawbyte( ) and pm_readlittleshort( ).

```
static int
GetByte(fp)
FILE    *fp;
{
    int    c;

    if ((c = getc(fp)) == EOF)
        pm_error("unexpected end of file" );

    return c;
}

static int
GetWord(fp)
FILE    *fp;
{
    int    c;

    c  = GetByte(fp);
    c |= (GetByte(fp) << 8);
    return c;
}
```

## 12.2 WRITING PCX FILES

Writing PCX files is somewhat simpler than reading them because we don't have to support every possible format, only a few common formats, which are adequate for all images of up to 8 bits per pixel:

- Single-bit 1-, 2-, and 4-plane formats, for images with up to 2, 4, or 16 colors.
- Eight-bit single-plane format, for images with up to 256 colors.

The conversion program starts with the usual macros and definitions. The image will be stored into the array pointed to by pixels[ ][ ], using the color hash table cht to map colors from true colors to color-map entries.

```
/* ppmtopcx.c - read a portable pixmap and produce a PCX file
*/
```

```
#include <stdio.h>
#include "ppm.h"
#include "ppmcmap.h"

#define MAXCOLORS       256
#define MAXPLANES       4

/*
 * Pointer to function returning an int
 */
typedef void (* vfunptr) ARGS(( int, int, unsigned char*, int,
                                int ));

static void PCXEncode ARGS(( FILE* fp, int GWidth, int GHeight,
int Colors, int Red[], int Green[], int Blue[],
vfunptr GetPlanes ));
static void PutPlane ARGS(( FILE* fp, unsigned char* buf,
                            int Size ));
static void ReadPlanes ARGS(( int y, int width, unsigned char* buf,
int planes, int bits ));
static void Putword ARGS(( int w, FILE* fp ));
static void Putbyte ARGS(( int b, FILE* fp ));

static pixel** pixels;
static colorhash_table cht;
```

First, the main program reads in the PPM image to convert.

```
void
main( argc, argv )
    int argc;
    char* argv[];
{
FILE* ifp;
int argn, rows, cols, colors, i;
pixval maxval;
pixel black_pixel;
colorhist_vector chv;
int Red[MAXCOLORS], Green[MAXCOLORS], Blue[MAXCOLORS];
char* usage = "[ppmfile]";

ppm_init( &argc, argv );

argn = 1;

if ( argn < argc )
    {
    ifp = pm_openr( argv[argn] );
```

```
        ++argn;
        }
    else
        ifp = stdin;

    if ( argn != argc )
        pm_usage( usage );

    pixels = ppm_readppm( ifp, &cols, &rows, &maxval );

    pm_close( ifp );
```

Since PCX images are almost invariably color-mapped, we use the color-map routines discussed in Chapter 9 to count the colors in the image. If there are more than 256 colors, we give up. (The user can use a separate color-quantizing program to reduce the number of colors.)

```
    /* Figure out the colormap. */
    pm_message( "computing colormap..." );
    chv = ppm_computecolorhist( pixels, cols, rows, MAXCOLORS,
    &colors );
    if ( chv == (colorhist_vector) 0 )
        pm_error(
            "too many colors, limit of %d", MAXCOLORS );
    pm_message( "%d colors found", colors );
```

Some applications that read PCX files use color-map entry 0 as a background color for the image. Black is usually a good background color, so we use ppm_addtocolorhist( ) to force a black entry into entry 0.

```
    /* Force black to slot 0 if possible. */
    PPM_ASSIGN(black_pixel, 0, 0, 0 );
    ppm_addtocolorhist(chv, &colors, MAXCOLORS,
        &black_pixel, 0, 0 );
```

Color-map entries are 8 bits per component, yielding a maximum component value of 255. If the incoming colors use a larger or smaller color range, we rescale them to a maximum value of 255 and store the three sets of components into Red[ ], Green[ ], and Blue[ ].

```
/* Now turn the ppm colormap into the appropriate PCX colormap.
*/
    if ( maxval > 255 )
        pm_message(
            "maxval is not 255 - automatically rescaling colors" );
    for ( i = 0; i < colors; ++i )
        {
        if ( maxval == 255 )
```

```
            {
            Red[i] = PPM_GETR( chv[i].color );
            Green[i] = PPM_GETG( chv[i].color );
            Blue[i] = PPM_GETB( chv[i].color );
            }
        else
            {
            Red[i] = (int) PPM_GETR( chv[i].color ) *
            255 / maxval;
            Green[i] = (int) PPM_GETG( chv[i].color ) *
            255 / maxval;
            Blue[i] = (int) PPM_GETB( chv[i].color ) *
            255 / maxval;
            }
    }
```

With the color maps set, we make a color hash table to allow the map entry for each pixel to be easily computed, give back the color-map histogram, which we no longer need, and call PCXEncode( ) to make the PCX file.

```
    /* And make a hash table for fast lookup. */
    cht = ppm_colorhisttocolorhash( chv, colors );
    ppm_freecolorhist( chv );

    /* All set, let's do it. */
    PCXEncode( stdout, cols, rows, colors, Red, Green, Blue,
    ReadPlanes );

    exit( 0 );
} /* main */
```

## Encoding a PCX File

PCXEncode( ) is a general-purpose PCX file encoder. It takes as its arguments the file to write to (stdout in our case), the size of the image, the three component arrays of the color map, and a function to call to get the raw plane data for each image row.

Writing a PCX file is the converse of reading one:

- Write the header.
- Write the compressed image data.
- Write the color map, if the image uses more than 16 colors.

Our PCX encoder starts by determining which of the possible subformats to use.

```
/*
 * PCXENCODE.C    - PCX Image compression interface
 * PCXEncode( file, GHeight, GWidth, Colors, Red, Green, Blue,
 * GetPlanes ) Write PCX data to file, size CHeight by GWidth,
 * number of colors
 * Colors, color map in Red[], Blue[], and Green[], pixel plane
 * data fetched by (*GetPlanes)(...)
 */

#define TRUE 1
#define FALSE 0

/* public */

static void
PCXEncode(fp, GWidth, GHeight, Colors, Red, Green, Blue,
GetPlanes )
FILE* fp;
int GWidth, GHeight;
int Colors;
int Red[], Green[], Blue[];
vfunptr GetPlanes;
{
        int             BytesPerLine;
        int             Planes;
        int             BitsPerPixel;
        unsigned char   *buf;
        int             i;
        int             n;
        int             y;

        /*
         * select number of planes and number of bits
         * per pixel according to number of colors
         */
        /*
         * 16 colors or less are handled as 1 bit per pixel
         * with 1, 2, 3 or 4 color planes.
         * more than 16 colors are handled as 8 bits per pixel
         * with 1 plane
         */
        if (Colors > 16)
        {
                BitsPerPixel    = 8;
                Planes          = 1;
        }
        else
        {
                BitsPerPixel    = 1;
```

```
        if (Colors > 8)
                Planes = 4;
        else if (Colors > 4)
                Planes = 3;
        else if (Colors > 2)
                Planes = 2;
        else
                Planes = 1;
}
```

Now we write the PCX file header. Most header fields get either fixed values or one of the image parameters.

```
/*
 * Write the PCX header
 */
Putbyte( 0x0a, fp);     /* .PCX magic number        */
Putbyte( 0x05, fp);     /* PC Paintbrush version    */
Putbyte( 0x01, fp);     /* .PCX run length encoding */
Putbyte( BitsPerPixel, fp);     /* bits per pixel   */

Putword( 0, fp );               /* x1  - image left   */
Putword( 0, fp );               /* y1  - image top    */
Putword( GWidth-1, fp );        /* x2  - image right  */
Putword( GHeight-1, fp );       /* y2  - image bottom */

Putword( GWidth, fp );          /* horizontal resolution */
Putword( GHeight, fp );         /* vertical resolution   */
```

The header includes a 16-entry color map. Even if the image really uses a 256-entry map at the end of the file, we have to write this color map.

```
/*
 * Write out the Color Map for images with 16 colors or less
 */
n = (Colors <= 16) ? Colors : 16;
for (i = 0; i < n; ++i)
{
        Putbyte( Red[i], fp );
        Putbyte( Green[i], fp );
        Putbyte( Blue[i], fp );
}
for (; i < 16; ++i)     /* pad out map if <16 colors */
{
        Putbyte( 255, fp );
        Putbyte( 255, fp );
        Putbyte( 255, fp );
}
```

We finish the header with more image parameters and some padding to bring it to its defined length of 128 bytes.

```
Putbyte( 0, fp);          /* reserved byte   */

Putbyte( Planes, fp);     /* number of color planes  */

BytesPerLine    = ((GWidth * BitsPerPixel) + 7) / 8;
Putword( BytesPerLine, fp );
                    /* number of bytes per scanline */

Putword( 1, fp);           /*  palette info           */

for (i = 0; i < 58; ++i)  /*  fill to end of header */
        Putbyte( 0, fp );

buf = (unsigned char *)malloc( MAXPLANES * BytesPerLine );
```

Now we write out the image row by row. A call to the GetPlanes routine (passed as an argument by the caller) gets the image data for the row, separated into planes if need be, with the planes stored one after another in buf[ ]. Then for each plane we call PutPlane( ) to compress the plane data and write it to the file.

```
for (y = 0; y < GHeight; ++y)
{
        (*GetPlanes)(y, GWidth, buf, Planes, BitsPerPixel);

        for (i = 0; i < Planes; ++i)
        PutPlane(fp, &buf[i * BytesPerLine], BytesPerLine);
}
```

Finally, if the image uses more than 16 colors, we write the actual color map at the end of the file.

```
/*
 * color map for > 16 colors is at end of file
 */
if (Colors > 16)
{
        Putbyte( 0x0c, fp); /* magic for 256 colors */
        for (i = 0; i < Colors; ++i)
        {
                Putbyte( Red[i], fp );
                Putbyte( Green[i], fp );
                Putbyte( Blue[i], fp );
        }
        for (; i < MAXCOLORS; ++i) /* pad to 256 entries */
        {
                Putbyte( 255, fp );
```

```
                                        Putbyte( 255, fp );
                                        Putbyte( 255, fp );
                                }
                        }

                        fclose( fp );
} /* PCXEncode */
```

## Image Encoding Helper Routines

The trickiest part of encoding a PCX image is separating the pixel data into planes. The encoding routine uses a clever separation technique that works for images of up to 4 bits per pixel. Each group of 8 pixels contains the data for 1 byte of each separated plane. Since we are handling no more than 4 bits per pixel, the 8 pixels define 4 bytes of plane data. We can temporarily store the 4 plane bytes in the 4 bytes of a long datum, which makes it possible to create the 4 bytes of plane data in parallel by OR-ing together 8 longs representing the 8 pixels.

To provide the long values representing the pixels, we define an auxiliary two-dimensional array called PixMap. An entry PixMap[i][j] contains the appropriate long for a pixel, which is the *i*th in the 8-pixel group and contains pixel value *j*. For each 8-pixel group, the code need only OR together the eight appropriate entries from PixMap to get the 4 bytes of scan lines.

Here are the first 16 entries of PixMap, corresponding to the first pixel in a group. The rest of the entries are computed on the fly below by shifting these entries an appropriate number of bits for each of the other seven pixel positions.

```
static unsigned long PixMap[8][16] =
{
        0x00000000L,    0x00000080L,    0x00008000L,    0x00008080L,
        0x00800000L,    0x00800080L,    0x00808000L,    0x00808080L,
        0x80000000L,    0x80000080L,    0x80008000L,    0x80008080L,
        0x80800000L,    0x80800080L,    0x80808000L,    0x80808080L,
};
```

ReadPlanes( ) handles two separate cases: one plane with 8 bits per plane, which needs no bit separation, and 1 to 4 planes with 1 bit per plane.

The simple 1-plane, 8-bit code appears first.

```
static void
ReadPlanes(y, width, buf, planes, bits)
int             y;
int             width;
unsigned char   *buf;
int             planes;
int             bits;
{
```

```
static int      first_time = 1;
unsigned char   *plane0, *plane1, *plane2, *plane3;
int             i, j, x;

/*
 * 256 color, 1 plane, 8 bits per pixel
 * just copy mapped pixels, no bit separation needed
 */
if (planes == 1 && bits == 8)
{
        for (x = 0; x < width; ++x)
                buf[x] = ppm_lookupcolor( cht,
                &pixels[y][x] );
        return;
}
```

The first time through, we fill in the rest of PixMap by shifting the first set of values 1 to 7 bits to the right, corresponding to bit positions 1 to 7 in the resulting bytes of plane data.

```
/*
 * must be 16 colors or less, 4 planes or less, 1 bit
 * per pixel
 */
if (first_time)
{
        for (i = 1; i < 8; ++i)
                for (j = 0; j < 16; ++j)
                        PixMap[i][j] = PixMap[0][j]
                        >> i;
        first_time = 0;
}
```

At last we actually create the plane data. In each group of 8 pixels, we look up each pixel in the color hash table to find the color index to store, and OR the appropriate set of bits for the bit position and color into a temporary long value. Then we store the bytes of the temporary value into the output planes. This completes the pixel-to-plane conversion.

```
i = (width + 7) / 8; /* width of each plane in bytes */

plane0  = buf;      /* find position for each plane */
plane1  = plane0 + i;
plane2  = plane1 + i;
plane3  = plane2 + i;

i       = 0;
x       = 0;
```

```
while ( x < width )
{
    register unsigned long      t;

    t  = PixMap[0][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[1][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[2][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[3][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[4][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[5][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[6][ppm_lookupcolor( cht, &pixels[y][x++] )];
    t |= PixMap[7][ppm_lookupcolor( cht, &pixels[y][x++] )];

    plane0[i] = t;
    if(planes > 1)plane1[i] = t >> 8;
    if(planes > 2)plane2[i] = t >> 16;
    if(planes > 3)plane3[i] = t >> 24;
    i++;
  }
} /* ReadPlanes */
```

## Compressing Plane Data

The last stage before writing out the file data is to compress it. PutPlane( ) scans through a row of plane data looking for repeated bytes. If it finds a repeat or a byte with the high 2 bits set (which would otherwise look like a repeat count), it writes a repeated group. Otherwise it writes a single byte. This process repeats until the end of the row.

```
/* Compress and write plane data from buf[], length Size,
 * to file fp
 */
static void
PutPlane(fp, buf, Size)
FILE            *fp;
unsigned char   *buf;
int             Size;
{
        unsigned char   *end;
        int             c;
        int             previous;
        int             count;

        end     = buf + Size;

        previous = *buf++;  /* start with first data byte */
        count    = 1;
```

```
while (buf < end)
{
        c = *buf++;
        if (c == previous && count < 63)
        {
                ++count;      /* in a repeated run */
                continue;
        }

        if (count > 1 || (previous & 0xc0) == 0xc0)
        {
                count |= 0xc0; /* write repeat count */
                Putbyte ( count , fp );
        }
        Putbyte(previous, fp); /* data or repeated byte */
                previous = c;
                count    = 1;
        }

        /* flush byte or group at end of data */
        if (count > 1 || (previous & 0xc0) == 0xc0)
        {
                count |= 0xc0;
                Putbyte ( count , fp );
        }
        Putbyte(previous, fp);
} /* PutPlane */
```

We have two small auxiliary routines, Putword( ), which does the same thing as pm_writelittleshort( ), and Putbyte( ), which is essentially the same as putc.

```
/*
 * Write out a word to the PCX file
 */
static void
Putword( w, fp )
int w;
FILE *fp;
{
        putc( w & 0xff, fp );
        putc( (w / 256) & 0xff, fp );
}

/*
 * Write out a byte to the PCX file
 */
static void
Putbyte( b, fp )
int b;
```

```
FILE *fp;
{
        putc( b & 0xff, fp );
}
```

## 12.3 POSSIBLE EXTENSIONS

A new PCX format has recently been introduced that supports true color images with 24 bits per pixel. This format stores data by row, first a row of red component bytes, a row of green component bytes, and a row of blue component bytes, repeated for each row. The header indicates that the file has 3 planes and 8 bits per pixel. Extending the reading and writing code to handle this format is straightforward.

CHAPTER 13

# GEM IMG

## 13.1 IMG FILES

Digital Research (now a part of Novell) defined the IMG image format for use with their Gem windowing system, which is widely used on Atari computers and less so on IBM-compatible PCs. IMG files can in principle contain color images of any depth, but in practice nearly all IMG files are single-plane bitmaps, so that is all we cover here.

An IMG file consists of a header followed by data planes. In a multi-plane file, the planes are stored in the order R1, G1, B1, X1, R2, G2, B2, X2, . . . , with the X plane being a gray image, starting with the most significant bit in each pixel.

The compression scheme in IMG files handles repeated data rows, byte-aligned runs of black or white pixels and repeated 2-byte patterns. In practice, most IMG files are uncompressed or use only the repeated row compression. Our IMG writing routines, for simplicity, do no compression at all.

GEM originally ran on Motorola 68000 systems, so all IMG files use the Motorola big-endian format.

## 13.2 READING IMG FILES

```
/*
 * Convert a GEM .img file to a portable bitmap file.
 */

#include <assert.h>
#include "pbm.h"
```

These two structures define the header present at the beginning of each file and the 4-byte code that indicates a repeated scan line.

```
/*
 * File header structure
 */
struct header {
    short           version;/* Image file version */
    unsigned short  hlen;    /* Header length in bytes */
    unsigned short  planes;  /* Number of planes */
    unsigned short  patlen;  /* Pattern definition length (bytes) */
    unsigned short  pxlen;   /* Pixel height (microns) */
    unsigned short  pxht;    /* Pixel height (microns) */
    unsigned short  linewid; /* Scan line width (bytes) */
    unsigned short  nitems;  /* Number of scan line items */
};

/*
 * Scan item header
 */
struct line {
   char             d1, d2;      /* Should be 0 */
   char             d3;                /* Should be 0xff */
   char             count;       /* Repetition count */
};

char            pattern[256];
```

We start with a sanity check to be sure that the C compiler on the local machine didn't add extra padding to the header structure that would make the in-memory version different from the IMG disk header. Then we check for the optional "debug" argument and open the input file. (This program doesn't read from standard input, though there's no particular reason it couldn't be modified to do so.)

```
void
main(argc, argv)
        int              argc;
        char            *argv[];
{
        int              debug = 0;
        FILE            *f;
        struct header    hd;
        int              x;
        int              i, j, k, l;
        int              c, cc, linerep;
        int                 rows, cols;
        bit                 *bitrow;
        short                 *sp;

        pbm_init( &argc, argv );
```

```
/* Check if the compiler aligns structures the way
 * we want */
if ( sizeof( struct header ) != 16 )
        pm_error( "sizeof( struct header ) != 16 ???" );

if (argc > 1 && !strcmp(argv[1], "-d")) {
        argc--;
        argv[1] = argv[0];
        ++argv;
        ++debug;
}

if (argc != 2)
        pm_usage("[-d] <gemfile>");
f = pm_openr( argv[1] );
```

Now we read in the header, using pm_readbigshort( ) to be sure to read the header in big-endian format.

```
/* read header in byte-order independent way */
sp = (short*)&hd;

for( i = 0; i < (sizeof(hd)/2); i++ ) {
        if( pm_readbigshort(f, &sp[i]) )
                pm_error("header read");
}
```

The IMG header can be arbitrarily long, containing extra information that we don't use. One of the header words contains the actual header length, so we read and discard any extra header.

Next, optional debugging code reports what the header contained.

```
for ( i = sizeof(struct header); i < hd.hlen; ++i )
    (void) getc(f);

if (debug) {
        pm_message(
                "version %d, hlen %d, planes %d, patlen %d",
                 hd.version, hd.hlen, hd.planes, hd.patlen, 0);
        pm_message(
                "pxlen %d, pxht %d, linewid %d, nitems %d",
                hd.pxlen, hd.pxht, hd.linewid, hd.nitems, 0);
}
```

Since IMG files store the image a row at a time, we can construct the PBM file a row at a time, avoiding the need to buffer the entire image in memory. We compute cols, the width of the image in pixels (rounded up to a multiple of eight,

because that's what IMG files do), and rows, the number of rows, write the PBM header, and allocate a single-row buffer, bitrow[ ].

```
cols = hd.linewid % 8 == 0 ? hd.linewid :
       hd.linewid + 8 - hd.linewid % 8;
rows = hd.nitems;
pbm_writepbminit( stdout, cols, rows, 0 );
bitrow = pbm_allocrow( cols );
```

Now we read in the rows. There are three kinds of possible data:

- A hex 80 byte introduces a "bit string," a byte count followed by that amount of bytes of literal data.
- A pair of zero bytes is a line repeat, meaning that the entire following row of data is to be repeated several times. The two zeros are followed by FF and the repeat count.
- A zero byte followed by a non-zero byte is a pattern repeat. The non-zero byte is a repeat count, with the data to be repeated following the repeat count. The length of the pattern to repeat is taken from the file header. (All pattern repeats in a given file are the same length, typically 2 bytes.)
- Anything else is a run of all-zero or all-one bytes. The high bit of the byte is the value to fill with, 1 or 0, and the low 7 bits are the number of bytes of fill.

A switch inside a while loop decodes the data for each line.

```
for (i = 0; i < hd.nitems; ) {
    x = 0;
    linerep = 0;
    while (x < hd.linewid) {
switch (c = getc(f)) {
case 0x80:    /* Bit string */
    c = getc(f);    /* Byte count */
    if (debug)
        pm_message(
            "bit string of %d bytes",
            c );
    for (j = 0; j < c; ++j) {
        cc = getc(f);
        for (k = 0x80; k; k >>= 1) {
            bitrow[x] = (k & cc) ?
            PBM_BLACK : PBM_WHITE;
            ++x;
        }
    }
    break;
```

```
            case 0:         /* Pattern run */
                c = getc(f);    /* Repeat count */
                if (debug)
                    pm_message(
                        "pattern run of %d repetitions",
                        c );
                                /* line repeat */
                                if (c == 0) {
                                        c = getc(f);
                                        if (c != 0x00ff)
                    pm_error( "badly formed line repeat" );
                                        linerep = getc(f);
                                        break;

                                }
                fread(pattern, 1, hd.patlen, f);
                for (j = 0; j < c; ++j)
                    for (l = 0; l < hd.patlen; ++l)
                        for (k = 0x80; k; k >>= 1) {
                            bitrow[x] = (k &
                            pattern[l]) ?
                            PBM_BLACK : PBM_WHITE;
                            ++x;
                        }
                break;

            default:    /* Solid run */
                if (debug)
                    pm_message(
                        "solid run of %d bytes %s",
                        c & 0x7f,
                        c & 0x80 ? "on" : "off" );
                            /* each byte had 8 bits DSB */
                            l = (c & 0x80) ? PBM_BLACK : PBM_WHITE;
                            c = (c & 0x7f) * 8;
                            for (j = 0; j < c; ++j) {
                                    bitrow[x] = l;
                                    ++x;
                            }
                break;

            case EOF:   /* End of file */
                pm_error( "end of file reached" );

            }

        }
```

At the end of the line, the amount of data retrieved should exactly match up with the line boundary. Assuming no error, we write the line the appropriate number of times to the output and repeat until we have all the rows.

```
                if ( debug )
                        pm_message( "EOL" );
                if ( x != hd.linewid )
                        pm_error( "EOL beyond edge" );
                do {
                        pbm_writepbmrow( stdout, bitrow, cols, 0 );
                        ++i;
                } while (linerep--);
        } /* for rows */
        pm_close( f );
        pm_close( stdout );
        exit(0);
} /* main */
```

On little-endian systems, the routine byteswap( ) swaps the bytes in the 2-byte values in the header.

```
#ifdef LITTLE_ENDIAN
/* swap the bytes in n shorts */
static void
byteswap(data, n)
        unsigned short* data;
        int             n;
{
        register        i;
        char           *p, c;

        for (i = 0, p = (char *) data; i < n; ++i, ++p) {
                c = *p;
                *p = *(p + 1);
                *++p = c;
        }
}
#endif /*LITTLE_ENDIAN*/
```

## 13.3 WRITING IMG FILES

We have a very simple but adequate IMG writer. Starting with a PBM file, it writes out an IMG header, then writes out uncompressed bit data. Practically the only work involved is to pack the pixels eight to a byte.

Our program starts with a definition of the IMG header, the static buffer outrow[ ] where each row of data is built up.

```
/* pbmtogem.c - read a portable bitmap and produce a GEM .img file
*/

#include <stdio.h>
#include "pbm.h"

/*
 * File header structure
 */
struct header {
        short           version;/* Image file version */
        unsigned short  hlen;   /* Header length in bytes */
        unsigned short  planes; /* Number of planes */
        unsigned short  patlen; /* Pattern definition length
                                   (bytes) */
        unsigned short  pxlen;  /* Pixel height (microns) */
        unsigned short  pxht;   /* Pixel height (microns) */
        unsigned short  linewid;
                                /* Scan line width (pixels) */
        unsigned short  nitems;
                                /* Number of scan line items */
};

#define MAXCOL 60       /* 60 shorts = 960 pixels per row */
static unsigned short outrow[MAXCOL];

static void putinit ARGS(( struct header* hdr ));
static void putbit ARGS(( bit b ));
static void putrest ARGS(( void ));
static void putitem ARGS(( void ));
static void putrow ARGS(( void ));
```

We start, as usual, processing the optional file name argument and reading the PBM header. The code here limits the width of an image to 960 pixels. (There's no good reason for the limit except that most IMG readers can't handle files wider than that, so there's not much point in producing wider IMG files.) We allocate bitrow[ ], the buffer for each incoming PBM row.

```
void
main( argc, argv )
    int argc;
    char* argv[];
    {
    FILE* ifp;
    bit* bitrow;
    register bit* bP;
    int rows, cols, format, row, col, pad;
    struct header hd;
```

```
pbm_init( &argc, argv );

if ( argc > 2 )
    pm_usage( "[pbmfile]" );

if ( argc == 2 )
    ifp = pm_openr( argv[1] );
else
    ifp = stdin;

pbm_readpbminit( ifp, &cols, &rows, &format );

if (cols > MAXCOL * 16) /* HACK: should dynamically allocate */
    cols = MAXCOL * 16; /* the output columns */
bitrow = pbm_allocrow( cols );
```

Now we have all the information we need for the IMG header, so we fill it in and write it with putinit( ), below. The width of the output image is rounded up to a multiple of 16 pixels, the convention for IMG files.

```
hd.version = 1;      /* Image file version */
hd.hlen = 16;        /* Header length in bytes */
hd.planes = 1;       /* Number of planes */
hd.patlen = 2;       /* Pattern definition length (bytes) */
hd.pxlen = 372;      /* Pixel height (microns) */
hd.pxht = 372;       /* Pixel height (microns) */
hd.linewid = ((cols + 15) / 16) * 16;
                     /* Scan line width (pixels) */
hd.nitems = rows;    /* Number of scan line items */

pad = hd.linewid - cols;

putinit( &hd );
```

For each row in the image, we read in the row, output the pixels in the row with putitem( ), and if necessary pad the row to a 16-bit boundary.

```
for ( row = 0; row < rows; ++row )
    {
    pbm_readpbmrow( ifp, bitrow, cols, format );
    for ( col = 0, bP = bitrow; col < cols; ++col, ++bP )
        putbit( *bP );
    for ( col = 0; col < pad; ++col )
        putbit( 0 );
    }
pm_close( ifp );
```

```
putrest( );

exit( 0 );
} /* main */
```

## Bit Packing and Writing

A set of general-purpose bit-packing routines supports writing data to an IMG file. The routine putinit( ) writes the header and sets up the static state variables. Each bit is written with putbit( ), which adds a bit to the current 16-bit item. If the item is full, it calls putitem( ), which copies the current item into the buffer outrow[ ]. If outrow[ ] is full, putitem( ) in turn calls putrow( ), which writes out the buffered data as an uncompressed bit string in the file. Finally, putrest( ) flushes any unwritten bits with putitem( ), and any buffered data in outrow[ ] with putrow( ). IMG files do not allow a single bit string to cross a row boundary; the code limits each output string to outmax words, which happens to be the width of a row.

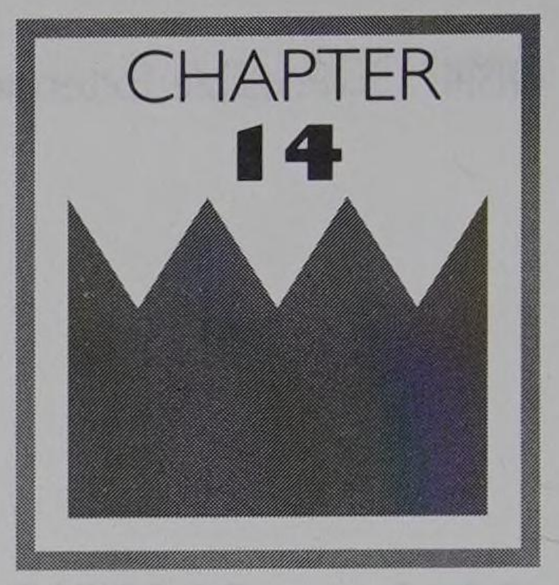
```
static short item,     /* current 16-bit group */
    outcol,            /* current output byte */
    outmax;            /* data bytes per row */
static short bitsperitem,
                       /* number of bits written to item so far */
    bitshift;          /* which bit number to write next */

/* write the IMG header and prepare to write data bits */
static void
putinit( hdr )
    struct header* hdr;
    {
    short *sp;
    int i;

    for(sp = (short*)hdr, i=0; i < 8; i++, sp++)
        pm_writebigshort(stdout, *sp); /* write big-endian data */

    item = 0;
    bitsperitem = 0;
    bitshift = 15;
    outcol = 0;
    outmax = hdr->linewid / 16;
    }

/* add a bit to the output image, store with putitem() if we
 * have a full 16 bit word
 */

#if __STDC__
```

```
static void
putbit( bit b )
#else /*__STDC__*/
static void
putbit( b )
    bit b;
#endif /*__STDC__*/
    {
    if ( bitsperitem == 16 )
        putitem( );
    ++bitsperitem;
    if ( b == PBM_BLACK )
        item += 1 << bitshift;
    --bitshift;
} /* putbit */

/* copy item to outrow[], write to file with putrow() if the row
 * is full
 */

static void
putitem( )
    {
    outrow[outcol++] = item;
    if (outcol >= outmax)
        putrow( );
    item = 0;
    bitsperitem = 0;
    bitshift = 15;
} /* putitem */

/* write a full row of data to the file as big-endian shorts */
static void
putrow( )
    {
    int i;

    (void) putc(0x80, stdout);          /* a Bit string */
    (void) putc(outcol*2, stdout);      /* count */
    for(i = 0; i < outcol; i++)
                            /* data as big-endian 16 bit words */
        pm_writebigshort(stdout, outrow[i]);
    outcol = 0;
} /* putrow */

/* write out any leftover buffered bits */
static void
putrest( )
    {
```

```
    if ( bitsperitem > 0 )
        putitem( );
    if ( outcol > 0 )
        putrow( );
} /* putrest */
```

## 13.4 POSSIBLE EXTENSIONS

Simple run-length compression code similar to that in the PCX encoder could make output files smaller. One could also borrow much of the code to handle multiplane images from PCX. In practice, though, IMG format is used only for small bitmaps such as icons, so the code we already have is sufficient.




# IFF/ILBM Bitmap Images

The Electronic Arts IFF format is an extremely flexible tagged file format. IFF files can contain anything from text to CAD files to sound clips to bitmaps. Here we treat just the ILBM subformat that handles pixel images. ILBM images can use any number of bits per pixel up to full 24-bit color and may or may not use a color map.

Each IFF file consists of a series of tagged chunks. A chunk is a 4-character tag, followed by a 4-byte length, followed by data. Reflecting the Motorola 68000 heritage of IFF, numbers are stored big-endian and all chunks are padded to 2-byte boundaries. An ILBM image is stored in a chunk of type "ILBM." Within that chunk are subchunks, in the same self-describing format. The most important subchunk types are:

**BMHD** The bitmap header that contains most of the image description.

**CAMG** A data word that is nominally specific to the screen hardware of the Commodore Amiga but in fact affects the interpretation of the image.

**CMAP** The color map, for color-mapped files.

**BODY** The actual bits of the image, stored by row, with each row stored by bit plane.

There are quite a few other subchunk types, most of which are specific to particular applications and can usually safely be ignored.

There are two special image interpretation modes coded in the CMAP word. One is "extra half-bright" mode, in which the high bit of each pixel is a flag that, if set, indicates that the pixel should be half the brightness it would otherwise be. We handle this by setting up a color map that makes the entries in the second half of the map the same as the first half, but with each component value divided by two. The other mode bit is Hold And Modify (HAM) mode, which completely changes the interpretation of pixel data. Rather than looking up each

pixel independently in the color map, in HAM mode each pixel differs from the pixel to its left in only one color component, and each pixel has a pair of flag bits saying which component to change. This allows a large set of colors to be encoded with relatively few bits, at the cost of some color smearing. HAM works well for digitized photographs or video frames where the smearing usually turns out to be unobtrusive.

## 14.1 ILBM HEADER

We start with the headerfile ilbm.h, which defines the structures and bits used in ILBM data chunks, along with the macro RowBytes( ), which computes the number of bytes an ILBM file uses for a given number of bits, taking into account the rounding to even byte boundaries.

```
/* ilbm.h - header file for IFF ILBM files */

#define RowBytes(cols)    ( ( ( (cols) + 15 ) / 16 ) * 2 )

/* definitions for BMHD */

typedef struct
    {
    unsigned short w, h;
    short x, y;
    unsigned char nPlanes, masking, compression, pad1;
    unsigned short transparentColor;
    unsigned char xAspect, yAspect;
    short pageWidth, pageHeight;
    } BitMapHeader;

#define mskNone                 0
#define mskHasMask              1
#define mskHasTransparentColor  2
#define mskLasso                3

#define cmpNone                 0
#define cmpByteRun1             1

/* definitions for CMAP */

typedef struct
    {
    unsigned char r, g, b;
    } ColorRegister;

/* definitions for CAMG */

#define         vmEXTRA_HALFBRITE       0x80
#define         vmHAM                   0x800
```

## 14.2 READING ILBM FILES

The ILBM file reader starts in the usual way, with declarations, PPM initialization, and opening the input file if need be.

```
/* ilbmtoppm.c - read an IFF ILBM file and produce a portable pixmap
*/

#include "ppm.h"
#include "ilbm.h"

static void getfourchars ARGS(( FILE* f, char fourchars[4] ));
static unsigned char get_byte ARGS(( FILE* f ));

void
main( argc, argv )
    int argc;
    char* argv[];
    {
    FILE* ifp;
    pixel* pixelrow;
    pixel* colormap = 0;
    int argn, colors, i, j, r, g, b, byte, bytes;
    short rows, cols = 0, row, col;
    int maxval;
    char iffid[5];
    unsigned char* body = 0;
    unsigned char* bp;
    unsigned char* ubp;
    unsigned char* rawrow;
    unsigned char* runbuf;
    unsigned char* Rrow;
    unsigned char* Grow;
    unsigned char* Brow;
    long formsize, bytesread, chunksize, viewportmodes = 0;
    int nPlanes, masking, compression, xAsp, yAsp, ham,
    hammask, allPlanes;

    ppm_init( &argc, argv );

    argn = 1;

    if ( argn < argc )
        {
        ifp = pm_openr( argv[argn] );
        argn++;
        }
    else
```

```
        ifp = stdin;

    if ( argn != argc )
        pm_usage( "[ilbmfile]" );
```

Each ILBM file must consist of a FORM chunk representing the whole file, containing an ILBM tag indicating that the file contains a pixel image.

```
    /* Read in the ILBM file. */
    iffid[4] = '\0';
    getfourchars( ifp, iffid );
    if ( strcmp( iffid, "FORM" ) != 0 )
        pm_error( "input is not a FORM type IFF file" );
    if ( pm_readbiglong( ifp, &formsize ) == -1 )
        pm_error( "EOF / read error" );
    getfourchars( ifp, iffid );
    if ( strcmp( iffid, "ILBM" ) != 0 )
        pm_error( "input is not an ILBM type FORM IFF file" );
    bytesread = 12;
```

Now we read the subchunks within the ILBM chunk, interpreting the ones we understand, until the entire ILBM form has been read.

```
    /* Main loop, parsing the IFF FORM. */
    while ( bytesread < formsize )
        {
        getfourchars( ifp, iffid );
        if ( pm_readbiglong( ifp, &chunksize ) == -1 )
            pm_error( "EOF / read error" );
        bytesread += 8;
```

The last subchunk in an ILBM image is supposed to be the BODY that contains the pixel data. We ignore any subchunks after that.

```
        if ( body != 0 )
            {
            pm_message(
                "\"%s\" chunk found after BODY chunk - skipping",
                iffid );
            for ( i = 0; i < chunksize; i++ )
                (void) get_byte( ifp );
            }
```

A BMHD chunk contains the number of rows and columns, number of planes (bits per pixel), and whether the data subchunk contains a plane of "masking" data, which we have to ignore. (When combining images, paint programs use the mask bits to decide which pixels to use and which to omit.)

```
else if ( strcmp( iffid, "BMHD" ) == 0 )
    {
    short junk;
    if ( pm_readbigshort( ifp, &cols ) == -1 )
        pm_error( "EOF / read error" );
    if ( pm_readbigshort( ifp, &rows ) == -1 )
        pm_error( "EOF / read error" );
    if ( pm_readbigshort( ifp, &junk ) == -1 )
        pm_error( "EOF / read error" );
    if ( pm_readbigshort( ifp, &junk ) == -1 )
        pm_error( "EOF / read error" );
    nPlanes = get_byte( ifp );
    masking = get_byte( ifp );
    compression = get_byte( ifp );
    (void) get_byte( ifp );                 /* pad1 */
    if ( pm_readbigshort( ifp, &junk ) == -1 )
                                            /* transparentColor */
        pm_error( "EOF / read error" );
    xAsp = get_byte( ifp );
    yAsp = get_byte( ifp );
    if ( pm_readbigshort( ifp, &junk ) == -1 )
                                            /* pageWidth */
        pm_error( "EOF / read error" );
    if ( pm_readbigshort( ifp, &junk ) == -1 )
                                            /* pageHeight */
        pm_error( "EOF / read error" );
    }
```

The CMAP subchunk contains the color map, stored as 3-byte RGB triples. Out of laziness, store the color map as a row of PPM pixels.

```
else if ( strcmp( iffid, "CMAP" ) == 0 )
    {
    colors = chunksize / 3;
    if ( colors > 0 )
        {
        colormap = ppm_allocrow( colors );
        for ( i = 0; i < colors; i++ )
            {
            r = get_byte( ifp );
            g = get_byte( ifp );
            b = get_byte( ifp );
            PPM_ASSIGN( colormap[i], r, g, b );
            }
        if ( colors * 3 != chunksize )
            (void) get_byte( ifp );
        }
    }
```

The CAMG subchunk contains one word of Amiga screen modes, which we save in case it specifies HAM or extra half-bright mode.

```
else if ( strcmp( iffid, "CAMG" ) == 0 )
    {
    if ( pm_readbiglong( ifp, &viewportmodes ) == -1 )
        pm_error( "EOF / read error" );
    }
```

We save the body data for later decoding.

```
else if ( strcmp( iffid, "BODY" ) == 0 )
    {
    body = (unsigned char*) malloc( chunksize );
    if ( body == 0 )
        pm_error( "out of memory" );
    if ( fread( body, 1, chunksize, ifp ) != chunksize )
        pm_error( "EOF / read error reading BODY chunk" );
    }
```

The following subchunk types can safely be ignored. Anything else provokes a warning and is also ignored.

```
else if ( strcmp( iffid, "GRAB" ) == 0 ||
          strcmp( iffid, "DEST" ) == 0 ||
          strcmp( iffid, "SPRT" ) == 0 ||
          strcmp( iffid, "CRNG" ) == 0 ||
          strcmp( iffid, "CCRT" ) == 0 ||
          strcmp( iffid, "DPPV" ) == 0 )
    {
    for ( i = 0; i < chunksize; i++ )
        (void) get_byte( ifp );
    }
else
    {
    pm_message( "unknown chunk type \"%s\" - skipping",
    iffid );
    for ( i = 0; i < chunksize; i++ )
        (void) get_byte( ifp );
    }

bytesread += chunksize;
}
```

Now we've read all the data from the file. We check to be sure we have all the information we need and warn the user if the file contains nonsquare pixels.

```
pm_close( ifp );
```

```
/* Done reading.  Now interpret what we got. */
if ( cols == 0 )
    pm_error( "no BMHD chunk found" );
if ( body == 0 )
    pm_error( "no BODY chunk found" );
if ( xAsp != yAsp )
    pm_message(
        "warning - nonsquare pixels; to fix do
                                a 'pnmscale -%cscale %g'",
        xAsp > yAsp ? 'x' : 'y',
        xAsp > yAsp ? (float) xAsp / yAsp :
                                (float) yAsp / xAsp );
```

We check for HAM mode. If it's set, the high 2 bits of each pixel are HAM control bits, and the rest of the bits are either a literal color component or a color table index. We compute maxval, the maximum component value, and rescale the color map appropriately.

```
    if ( ( viewportmodes & vmHAM ) && nPlanes != 24 )
        {
        ham = 1;
        hammask = ( 1 << ( nPlanes - 2 ) ) - 1;
        maxval = pm_bitstomaxval( nPlanes - 2 );
        if ( maxval > PPM_MAXMAXVAL )
        pm_error(
"nPlanes is too large - try reconfiguring with PGM_BIGGRAYS\n"
"    or without PPM_PACKCOLORS" );
        if ( colormap != 0 )
            for ( i = 0; i < colors; i++ )
                {
                r = PPM_GETR( colormap[i] ) >> ( 10 - nPlanes );
                g = PPM_GETG( colormap[i] ) >> ( 10 - nPlanes );
                b = PPM_GETB( colormap[i] ) >> ( 10 - nPlanes );
                PPM_ASSIGN( colormap[i], r, g, b );
                }
        }
    else
```

HAM is not set, so the image uses a conventional color map, or it is a 24-bit full-color image, or (if there is no color map and less than 24 planes) it is gray scale.

```
        {
        ham = 0;
        if ( colormap != 0 )
            maxval = 255;          /* colormap contains bytes */
        else if ( nPlanes == 24 )
            maxval = 255;
```

```
        else
            maxval = pm_bitstomaxval( nPlanes );
        if ( maxval > PPM_MAXMAXVAL )
            pm_error(
"nPlanes is too large - try reconfiguring with PGM_BIGGRAYS\n"
"    or without PPM_PACKCOLORS" );
        }
```

If the extra half-bright mode is set, we need to double the size of the color map since the color map stored in the file doesn't account for the high "half-bright" bit. The new upper half is the same as the lower half but with half the intensity for each component.

```
    if ( viewportmodes & vmEXTRA_HALFBRITE )
    {
    pixel* tempcolormap;

    tempcolormap = ppm_allocrow( colors * 2 );
    for ( i = 0; i < colors; i++ )
        {
        tempcolormap[i] = colormap[i];
        PPM_ASSIGN(
        tempcolormap[colors + i], PPM_GETR(colormap[i]) / 2,
        PPM_GETG(colormap[i]) / 2, PPM_GETB(colormap[i]) / 2 );
        }
    ppm_freerow( colormap );
    colormap = tempcolormap;
    colors *= 2;
    }
```

Now we prepare to extract the image from the bits in the BODY chunk. We determine the number of bit planes to expect in the file and prepare to write the output PPM file.

```
    if ( colormap == 0 && nPlanes != 24 )
        pm_message( "no colormap - interpreting values as
        grayscale" );
    allPlanes = nPlanes + ( masking == mskHasMask ? 1 : 0 );

    ppm_writeppminit( stdout, cols, rows, (pixval) maxval, 0 );
```

We will build each row of PPM pixels in pixelrow[ ]. For full-color images, we need a temporary array for each component, Rrow[ ], Grow[ ], and Brow[ ]. For other images, we use a single array rawrow[ ] for the assembled pixel map values. Images may use PackBits compression; if so, the compressed data will be expanded into rawrows[ ].

```
    pixelrow = ppm_allocrow( cols );
```

```
if ( nPlanes == 24 )
    {
    Rrow = (unsigned char*) malloc( cols );
    Grow = (unsigned char*) malloc( cols );
    Brow = (unsigned char*) malloc( cols );
    if ( Rrow == 0 || Grow == 0 || Brow == 0 )
        pm_error( "out of memory" );
    }
else
    {
    rawrow = (unsigned char*) malloc( cols );
    if ( rawrow == 0 )
        pm_error( "out of memory" );
    }
runbuf = (unsigned char*) malloc( RowBytes( cols ) );
if ( runbuf == 0 )
    pm_error( "out of memory" );
```

Now we decode the image, a row at a time. For each row, we start by zeroing the buffer where the pixels will be assembled a plane at a time.

```
bp = body;
for ( row = 0; row < rows; row++ )
    {
    /* Extract rawrow from the image. */
    if ( nPlanes == 24 )
        for ( col = 0; col < cols; ++col )
            Rrow[col] = Grow[col] = Brow[col] = 0;
    else
        for ( col = 0; col < cols; ++col )
            rawrow[col] = 0;
```

We iterate over the planes. If the body is compressed, we expand the row into rawrow[ ]. If not, we can use the plane data directly from the file.

```
    for ( i = 0; i < allPlanes; i++ )
        {
        switch ( compression )
            {
            case cmpNone:
            ubp = bp;   /* use incoming data directly */
            bp += RowBytes( cols );
            break;

            case cmpByteRun1:
            ubp = runbuf;
            bytes = RowBytes( cols );
            do
                {
```

```
            byte = *bp++;
            if ( byte <= 127 )
                for ( j = byte, bytes -= j + 1; j >= 0;
                j-- )
                    *ubp++ = *bp++;
            else if ( byte != 128 )
                for ( j = 256 - byte, bytes -
                = j + 1,
                byte = *bp++;
                      j >= 0; j-- )
                    *ubp++ = byte;
            }
        while ( bytes > 0 );
        if ( bytes < 0 )
            pm_error( "error doing ByteRun
                                    decompression" );
        ubp = runbuf;
        break;

        default:
        pm_error( "unknown compression type" );
        }
```

The mask plane, if any, is stored after the rest of the image, so we can just ignore it.

```
    if ( i >= nPlanes )
        continue;          /* ignore mask plane */
```

Full-color image rows are stored as 8 planes of red, 8 planes of blue, and 8 planes of green data. We OR in the plane data to the appropriate place in the appropriate component.

```
    if ( nPlanes == 24 )
        {
        for ( col = 0; col < cols; col++ )
            if ( i < 8 )
                { /* red */
                if ( ubp[col / 8] & ( 128 >> ( col % 8 ) ) )
                    Rrow[col] |= 1 << i;
                }
            else if ( i > 15 )
                { /* blue */
                if ( ubp[col / 8] & ( 128 >> ( col % 8 ) ) )
                    Brow[col] |= 1 << (i-16);
                }
            else
                { /* green */
                if ( ubp[col / 8] & ( 128 >> ( col % 8 ) ) )
```

```
                    Grow[col] |= 1 << (i-8);
                }
            }
```

Other images are, for now, just planes of data. We build the pixels in rawrow[ ].

```
    else
        for ( col = 0; col < cols; col++ )
            if ( ubp[col / 8] & ( 128 >> ( col % 8 ) ) )
                rawrow[col] |= 1 << i;
    }
```

Now that we have assembled a row of pixels from the file, we have to interpret the pixels.

If the file is a HAM file, we now interpret the peculiar HAM coding. For each pixel, the high 2 bits control the interpretation. If those bits are 0, the lower bits specify a grayscale color-map entry; if non-zero, they specify a color-map entry to replace one of the components of the previous pixel.

```
/* Interpret rawrow into pixels. */
r = g = b = 0;
for ( col = 0; col < cols; col++ )
    if ( ham )
        { /* HAM mode. */
        switch ( ( rawrow[col] >> nPlanes - 2 ) & 0x3 )
            {
            case 0:
            if ( colormap != 0 && colors >= maxval )
                pixelrow[col] = colormap[rawrow[col]
                & hammask];
            else
                PPM_ASSIGN(
                    pixelrow[col], rawrow[col] & hammask,
                    rawrow[col] & hammask, rawrow[col]
                    & hammask );
            r = PPM_GETR( pixelrow[col] );
            g = PPM_GETG( pixelrow[col] );
            b = PPM_GETB( pixelrow[col] );
            break;

            case 1:
            b = rawrow[col] & hammask;
            PPM_ASSIGN( pixelrow[col], r, g, b );
            break;

            case 2:
            r = rawrow[col] & hammask;
            PPM_ASSIGN( pixelrow[col], r, g, b );
            break;
```

```
                case 3:
                g = rawrow[col] & hammask;
                PPM_ASSIGN( pixelrow[col], r, g, b );
                break;

                default:
                pm_error( "impossible HAM code" );
                }
            }
```

For non-HAM images, full-color pixels are copied directly to the output, color-mapped pixels are looked up in color map[ ], and non-color-mapped pixels (which almost never occurs) are interpreted as grayscale, with the same value for each component.

This completes the translation of each row, so we write it out with ppm_writeppmrow( ) and loop until done.

```
            else if ( nPlanes == 24 )
                /* 24-bit image. */
                PPM_ASSIGN( pixelrow[col], Rrow[col],
                Grow[col],
                Brow[col] );
            else if ( colormap != 0 )
                /* Non-HAM colormapped. */
                pixelrow[col] = colormap[rawrow[col]];
            else
                /* Non-HAM direct - weird. */
                PPM_ASSIGN(
                    pixelrow[col], rawrow[col], rawrow[col],
                    rawrow[col] );

        /* And write out the row. */
        ppm_writeppmrow( stdout, pixelrow, cols, (pixval)
            maxval,
            0 );
            }

        pm_close( stdout );

        exit( 0 );
        }
```

## Reader Auxiliary Routines

Two simple routines support the ILBM reader. The first, get_byte( ), is just getc( ) with an error check. The second, getfourchars( ), reads the four bytes of an IFF header into a string.

```
static unsigned char
get_byte( f )
    FILE* f;
    {
    int i;

    i = getc( f );
    if ( i == EOF )
        pm_error( "EOF / read error" );

    return (unsigned char) i;
    }

/* read a four byte IFF header into a string  */
static void
getfourchars( f, fourchars )
    FILE* f;
    char fourchars[4];
    {
    fourchars[0] = get_byte( f );
    fourchars[1] = get_byte( f );
    fourchars[2] = get_byte( f );
    fourchars[3] = get_byte( f );
} /* getfourchars */
```

## 14.3 WRITING ILBM FILES

The ILBM writing code supports two general writing strategies—regular and HAM. If an image has few enough colors (no more than 32) to be written as a regular ILBM image, it does so. Otherwise it writes an approximate HAM image. The user must specify –hamif to confirm that a HAM image is appropriate. These choices reflect the capabilities of the Amiga systems that most often display ILBM images.

We start by checking the flags, opening the PPM file, reading in the whole thing, and computing the color map.

```
/* ppmtoilbm.c - read a portable pixmap and produce an IFF ILBM file
*/

#include "ppm.h"
#include "ppmcmap.h"
#include "ilbm.h"

#define MAX_COLORS 256
#define MAX_HAMIF_COLORS 32
```

```
static int colorstobpp ARGS(( int colors ));
static void put_chunk_head ARGS(( char* str, long size ));
static void put_fourchars ARGS(( char* str ));
static void put_big_short ARGS(( short s ));
static void put_big_long ARGS(( long l ));
static void put_byte ARGS(( unsigned char b ));

void
main( argc, argv )
    int argc;
    char* argv[];
    {
    FILE* ifp;
    int argn, rows, cols, colors, i, hamif, ham, nPlanes,
    formsize, cmapsize;
    pixel** pixels;
    pixel* pP;
    int row, col, plane;
    pixval maxval;
    colorhash_table cht;
    colorhist_vector chv;
    unsigned char* raw_rowbuf;
    unsigned char* coded_rowbuf;
    char* usage = "[-hamif] [ppmfile]";

    ppm_init( &argc, argv );

    argn = 1;
    hamif = 0;

    if ( argn < argc && argv[argn][0] == '-' &&
                                    argv[argn][1] != '\0' )
        {
        if ( pm_keymatch( argv[argn], "-hamif", 2 ) )
            hamif = 1;
        else if ( pm_keymatch( argv[argn], "-nohamif", 2 ) )
            hamif = 0;
        else
            pm_usage( usage );
        ++argn;
        }

    if ( argn < argc )
        {
        ifp = pm_openr( argv[argn] );
        ++argn;
        }
    else
        ifp = stdin;
```

```
if ( argn != argc )
    pm_usage( usage );

pixels = ppm_readppm( ifp, &cols, &rows, &maxval );

pm_close( ifp );

/* Figure out the colormap. */
pm_message( "computing colormap..." );
chv = ppm_computecolorhist( pixels, cols, rows,
MAX_COLORS, &colors );
```

If the color map is too large to compute, or if it has more than 32 (MAX_HAMIF_COLORS), we have to produce a HAM file. If not, we make a color hash table for later use.

```
if ( hamif && chv == (colorhist_vector) 0 )
{
pm_message(
    "too many colors - proceeding to write a HAM file" );
    pm_message(
        "if you want a non-HAM file, try doing a 'ppmquant %d'",
        MAX_HAMIF_COLORS );
    ham = 1;
    colors = 16;
    nPlanes = 6;
}
else if ( chv == (colorhist_vector) 0 )
pm_error( "too many colors - try doing a ppmquant" );
else if ( hamif && colors > MAX_HAMIF_COLORS )
{
pm_message(
  "%d colors found - proceeding to write a HAM file", colors );
pm_message(
    "if you want a non-HAM file, try doing a 'ppmquant %d'",
    MAX_HAMIF_COLORS );
ham = 1;
colors = 16;
nPlanes = 6;
}
else
{
pm_message( "%d colors found", colors );
ham = 0;
nPlanes = colorstobpp( colors );
/* Make a hash table for fast color lookup. */
cht = ppm_colorhisttocolorhash( chv, colors );
}
```

Now we have all of the information needed to write the headers, so we write the FORM ILBM header, calculating the size of the entire file as needed for the chunk header.

```
/* Start off the ILBM with a FORM ILBM. */
cmapsize = colors * 3;
if ( odd( colors ) )
    cmapsize++;                         /* pad CMAP to word */
formsize =
    4 +                                 /* ILBM */
    4 + 4 + 20 +                         /* BMHD size header*/
    4 + 4 + cmapsize +                  /* CMAP size colors */
    4 + 4 + 4 +                          /* CAMG size val */
    4 + 4 + rows * nPlanes * RowBytes(cols);
                                        /* BODY size data */
put_chunk_head( "FORM", formsize );
put_fourchars( "ILBM" );
```

The first subchunk is the BMHD bitmap headers. All values except for the image size have fixed values.

```
/* Write out the BMHD. */
put_chunk_head( "BMHD", 20 );
put_big_short( cols );
put_big_short( rows );
put_big_short( 0 );                         /* x */
put_big_short( 0 );                         /* y */
put_byte( nPlanes );
put_byte( mskNone );
put_byte( cmpNone );
put_byte( 0 );                          /* pad1 */
put_big_short( 0 );                         /* transparentColor */
put_byte( 10 );                          /* xAsp */
put_byte( 10 );                          /* yAsp */
put_big_short( cols );                  /* pageWidth */
put_big_short( rows );                  /* pageHeight */
```

We always write a color map. If the image is color-mapped, we write the real color map, scaling it if necessary. For a HAM file, we write a simple color map that provides 16 levels of gray.

```
/* Write out the CMAP. */
if ( maxval > 255 )
    pm_message( "maxval is not 255 - automatically
                                        rescaling colors" );
put_chunk_head( "CMAP", cmapsize );
for ( i = 0; i < colors; i++ )
    {
    if ( ham )
```

```
            { /* Grayscale colormap for HAM. */
            put_byte( i << 4 );
            put_byte( i << 4 );
            put_byte( i << 4 );
            }
        else if ( maxval == 255 )
            {
            put_byte( PPM_GETR( chv[i].color ) );
            put_byte( PPM_GETG( chv[i].color ) );
            put_byte( PPM_GETB( chv[i].color ) );
            }
        else
            {
            put_byte( (char) ( (int) PPM_GETR(chv[i].color) *
                    255 / maxval ) );
            put_byte( (char) ( (int) PPM_GETG(chv[i].color) *
                    255 / maxval ) );
            put_byte( (char) ( (int) PPM_GETB(chv[i].color) *
                    255 / maxval ) );
            }
        }
    if ( odd( colors ) )
        put_byte( 0 );                    /* Pad to word. */
```

The CAMG subchunk records whether this is a HAM file.

```
/* Write out the CAMG. */
put_chunk_head( "CAMG", 4 );
if ( ham )
    put_big_long( vmHAM );
else
    put_big_long( 0 );
```

The final subchunk is the BODY, the actual data. We allocate two buffers, raw_rowbuf[ ], which holds the list of raw encoded pixel values for a row, and coded_rowbuf[ ], which holds a plane of bits as the encoded pixels are stored a plane at a time.

```
/* And finally, encode and write out the BODY. */
raw_rowbuf = (unsigned char *) malloc( cols );
coded_rowbuf = (unsigned char *) malloc( RowBytes(cols) );
if ( raw_rowbuf == 0 || coded_rowbuf == 0 )
    pm_error( "out of memory" );
put_chunk_head( "BODY", rows * nPlanes * RowBytes(cols) );
for ( row = 0; row < rows; row++ )
    {
```

If we're writing a HAM image, we have to compute the best approximation to each pixel possible using HAM encoding. For the first pixel in the line, this

is a grayscale approximation of the pixel's luminance. For the rest of the pixels in the line, we compute four error "distances," measures of the representation of the pixel for the four possible HAM codes we could output. For example, if we output a red code, the blue and green components would still have their previous values, so the red distance, called dred, is the sum of the errors in the blue and green components. The value of dgray is the error that results if we use a zero HAM code to insert a gray pixel. We use the HAM code for whichever of the four possibilities has the smallest error distance.

```
/* Go from pixels to raw bytes. */
if ( ham )
    { /* HAM mode. */
    register int noprev, pr, pg, pb, r, g, b, l;
    noprev = 1;
    for ( col = 0, pP = pixels[row]; col < cols; col++, pP++ )
    {
    r = PPM_GETR( *pP );
    g = PPM_GETG( *pP );
    b = PPM_GETB( *pP );
    l = PPM_LUMIN( *pP );
    if ( maxval != 15 )
        {
        r = r * 15 / maxval;
        g = g * 15 / maxval;
        b = b * 15 / maxval;
        l = l * 15 / maxval;
        }
    if ( noprev )
        { /* No previous pixels, must use the gray option. */
        raw_rowbuf[col] = l;
        pr = pg = pb = l;
        noprev = 0;
        }
    else
        {
        register int dred, dgreen, dblue, dgray;

        /* Compute distances for the four options. */
        dred = abs( g - pg ) + abs( b - pb );
        dgreen = abs( r - pr ) + abs( b - pb );
        dblue = abs( r - pr ) + abs( g - pg );
        dgray = abs( r - l ) + abs( g - l ) + abs( b - l );

        if ( dgray < dred && dgray < dgreen && dgray < dblue )
        {
        raw_rowbuf[col] = l;
        pr = pg = pb = l;
        }
```

```
            else if ( dblue < dred && dblue < dgreen )
            {
            raw_rowbuf[col] = ( 1 << 4 ) + b;
            pb = b;
            }
            else if ( dred < dgreen )
            {
            raw_rowbuf[col] = ( 2 << 4 ) + r;
            pr = r;
            }
            else
            {
            raw_rowbuf[col] = ( 3 << 4 ) + g;
            pg = g;
            }
            }
        }
        }
```

Non-HAM pixels are considerably easier. All we have to do is to find the appropriate color-map index for the pixel.

```
        else
            /* Non-HAM. */
            for ( col = 0, pP = pixels[row]; col < cols; col++,
            pP++ )
                raw_rowbuf[col] = ppm_lookupcolor( cht, pP );
```

We write the encoded row in raw_rowbuf[ ] a plane at a time, starting with the low-order plane, i.e., the low-order bit in each encoded pixel.

When we have finished writing all of the planes for all of the rows, we're done.

```
        /* Encode and write raw bytes in plane-interleaved form. */
    for ( plane = 0; plane < nPlanes; plane++ )
        {
        int mask;
        unsigned char *cp;

        mask = 1 << plane;
        cp = coded_rowbuf;
        *cp = 0;
        for ( col = 0; col < cols; col++ )
            {
            int b;

            b = ( raw_rowbuf[col] & mask ) ? 1 : 0;
            *cp |= b << ( 7 - col % 8 );
            if ( col % 8 == 7 )
                {
```

```
                    cp++;
                    *cp = 0;
                    }
                }
            fwrite( coded_rowbuf, RowBytes(cols), 1, stdout );
            } /* for planes */
        } /* for image rows */

    exit( 0 );
    }
```

## ILBM Writing Auxiliary Routines

A small set of auxiliary routines supports writing ILBM files. The number of pixel bits needed for a given number of colors is determined by colorstobpp( ), and a few output routines write IFF headers, code words, numbers, and bytes.

```
/* compute log 2 of number of colors, number of planes needed per
 * pixel
 */
static int
colorstobpp( colors )
    int colors;
    {
    int bpp;

    if ( colors <= 2 )
        bpp = 1;
    else if ( colors <= 4 )
        bpp = 2;
    else if ( colors <= 8 )
        bpp = 3;
    else if ( colors <= 16 )
        bpp = 4;
    else if ( colors <= 32 )
        bpp = 5;
    else if ( colors <= 64 )
        bpp = 6;
    else if ( colors <= 128 )
        bpp = 7;
    else if ( colors <= 256 )
        bpp = 8;
    else
        pm_error( "can't happen" );

    return bpp;
    }

/* write IFF chunk header, code and size */
```

```
static void
put_chunk_head( str, size )
    char* str;
    long size;
    {
    put_fourchars( str );
    put_big_long( size );
    }

/* write four-letter CODE */
static void
put_fourchars( str )
char* str;
    {
    fputs( str, stdout );
    }

/* write big-endian short, give up on error */
#if __STDC__
static void
put_big_short( short s )
#else /*__STDC__*/
static void
put_big_short( s )
    short s;
#endif /*__STDC__*/
    {
    if ( pm_writebigshort( stdout, s ) == -1 )
        pm_error( "write error" );
    }

/* write big-endian long, give up on error */
#if __STDC__
static void
put_big_long( long l )
#else /*__STDC__*/
static void
put_big_long( l )
    long l;
#endif /*__STDC__*/
    {
    if ( pm_writebiglong( stdout, l ) == -1 )
        pm_error( "write error" );
    }

/* write 1 byte to output file, same as putchar() */
#if __STDC__
static void
put_byte( unsigned char b )
```

```
#else /*__STDC__*/
static void
put_byte( b )
    unsigned char b;
#endif /*__STDC__*/
    {
    (void) putc( b, stdout );
    }
```

# Uncompressed Formats

CHAPTER 15

# Windows Bitmap

Microsoft Windows defines its own bitmap file format, used most often for little images to be incorporated into icons or screen backgrounds. The format supports 1, 4, or 8 bit per pixel color-mapped images and, in recent versions, 24-bit true-color images. Images can either be uncompressed or compressed using a run-length technique similar to PackBits. In practice, few BMP files are compressed and we do not discuss compression here.

MS Windows has a conventional 16-color color map that it uses on the screen, and it approximates the colors in a displayed BMP file using that map. (The map includes black, white, and dim and bright versions of red, green, yellow, blue, magenta, purple, and gray.) Naturally, the closer an image's color map is to those colors, the better it will look when Windows displays it. On screens that allow 256 colors, Windows only installs its color map into the first 16 entries, so the other 240 entries can hold the actual colors in a file.

A BMP file consists of a fixed-length header, followed by the color map, if present, followed by the image stored pixel by pixel, packed into bytes in a straightforward way.

## 15.1 COMMON BMP HEADER

The include file bmp.h defines the structure of the fixed header of a BMP file.

```
/*
**  bmp.h    BMP file structure
*/

/* 1, 2, and 4 bytes fields in BMP files */
typedef unsigned char bmpbyte;
typedef unsigned short bmpshort;
typedef unsigned long bmplong;
```

```
struct BmpHeader {                      /* start of BITMAPFILEHEADER */
        bmpbyte magic1, magic2;
        bmplong filesize;
        bmpshort res1, res2;
        bmplong pixeloffset;

        bmplong bmisize;            /* start of BITMAPINFOHEADER */
        bmplong cols;
        bmplong rows;
        bmpshort planes;
        bmpshort bitsperpixel;
        bmplong compression;
        bmplong cmpsize;            /* size of compressed image */
        bmplong xscale;                     /* pixels per meter */
        bmplong yscale;
        bmplong colors;                   /* number of colors used */
        bmplong impcolors;          /* number of important colors */
        /* color map starts here */
};

typedef struct BmpHeader BMPHEADER;

#define         TRUE        1
#define         FALSE        0
```

## 15.2 READING BMP FILES

The BMP reader starts in the usual way by checking flags, in this case just a verbose option, and opening the file.

```
/* bmptoppm.c
** Microsoft BMP to PPM files
*/

#include "ppm.h"
#include "ppmcmap.h"
#include "bmp.h"

static char usage[] = "[-verbose] [bmpfile]";
static int verbose = FALSE;

static void readBMP ARGS(( FILE *fd ));

void
main( argc, argv )
int argc;
char* argv[];
{
```

```
    int argn = 1;
    FILE *in;

    ppm_init(&argc, argv);

    while (argn < argc && argv[argn][0] == '-'
                       && argv[argn][1] != '\0') {
        if (pm_keymatch(argv[argn], "-verbose", 2)) {
            verbose = TRUE;
        } else {
            pm_usage(usage);
        }
        ++argn;
    }
    if (argn != argc) {
        in = pm_openr(argv[argn]);
        ++argn;
    } else {
        in = stdin;
#ifdef MSDOS
        /* be sure to read standard input as binary */
        _setmode( _fileno(stdin), _O_BINARY);
#endif /* MSDOS */
    }

    if (argn != argc)
        pm_usage(usage);
```

All the work is done in readBMP( ), which we examine next.

```
    readBMP(in);
    pm_close(in);
    pm_close(stdout);
    exit(0);
}
```

## Decoding BMP Files

We start by reading the 2-byte header and ensuring that the file starts with the signature BM. Then some rather ugly code reads in all of the rest of the fields in the header using byte-order independent library routines. If the user requested, we report the contents of the header.

```
void
readBMP(fd)
FILE *fd;
{
    struct BmpHeader bh;
    pixel *colormap = NULL;
```

```
    pixel **image;
    int bytesread;
    int row, col;

    /* read the header word by word to avoid byte order problems */

    bh.magic1 = pbm_getrawbyte(fd);
    bh.magic2 = pbm_getrawbyte(fd);

    if(bh.magic1 != 'B' || bh.magic2 != 'M')
        pm_error("Not a BMP file");

    if( pm_readlittlelong(fd, (long*)&bh.filesize)
        || pm_readlittleshort(fd, (short*)&bh.res1)
        || pm_readlittleshort(fd, (short*)&bh.res2)
        || pm_readlittlelong(fd, (long*)&bh.pixeloffset)
        || pm_readlittlelong(fd, (long*)&bh.bmisize)
        || pm_readlittlelong(fd, (long*)&bh.cols)
        || pm_readlittlelong(fd, (long*)&bh.rows)
        || pm_readlittleshort(fd, (short*)&bh.planes)
        || pm_readlittleshort(fd, (short*)&bh.bitsperpixel)
        || pm_readlittlelong(fd, (long*)&bh.compression)
        || pm_readlittlelong(fd, (long*)&bh.cmpsize)
        || pm_readlittlelong(fd, (long*)&bh.xscale)
        || pm_readlittlelong(fd, (long*)&bh.yscale)
        || pm_readlittlelong(fd, (long*)&bh.colors)
        || pm_readlittlelong(fd, (long*)&bh.impcolors) )
        pm_error("EOF in header");

    if(verbose) {
        fprintf(stderr, "filesize %d\n", bh.filesize);
        fprintf(stderr, "res1 %d\n", bh.res1);
        fprintf(stderr, "res2 %d\n", bh.res2);
        fprintf(stderr, "pixeloffset %d\n", bh.pixeloffset);
        fprintf(stderr, "bmisize %d\n", bh.bmisize);
        fprintf(stderr, "cols %d\n", bh.cols);
        fprintf(stderr, "rows %d\n", bh.rows);
        fprintf(stderr, "planes %d\n", bh.planes);
        fprintf(stderr, "bitsperpixel %d\n", bh.bitsperpixel);
        fprintf(stderr, "compression %d\n", bh.compression);
        fprintf(stderr, "cmpsize %d\n", bh.cmpsize);
        fprintf(stderr, "xscale %d\n", bh.xscale);
        fprintf(stderr, "yscale %d\n", bh.yscale);
        fprintf(stderr, "colors %d\n", bh.colors);
        fprintf(stderr, "impcolors %d\n", bh.impcolors);
    }
    if(bh.compression)
        pm_error("Can't handle compressed images, sorry\n");
```

The value of the colors field is often left 0; in that case we assume the image uses the maximum number of colors that can be represented in the number of bits used.

```
if(bh.colors == 0)
    bh.colors = 1<<bh.bitsperpixel;

bytesread = 54;     /* file position so far */
```

For images with 1, 4, or 8 bits per pixel, a color map follows the header. Each color-map entry consists of the usual three components, with an as yet unused fourth component that we ignore. We store the color map in a PPM pixel vector.

```
switch(bh.bitsperpixel) {
    int i;

case 24:
    break;  /* no color map for true color image */

case 1: case 4: case 8:
    /* need to read a color map */
    colormap = ppm_allocrow(bh.colors);
                                /* use ppm pixel vector */
    if(!colormap)
        pm_error("Out of memory");

    for(i = 0; i < bh.colors; i++) {
        int r, g, b;

        b = pbm_getrawbyte(fd);
        g = pbm_getrawbyte(fd);
        r = pbm_getrawbyte(fd);
        (void)pbm_getrawbyte(fd);   /* spare byte */

        if(verbose)
            fprintf(stderr," %2d:
                    (%3d,%3d,%3d)\n", i, r, g, b);

        PPM_ASSIGN(colormap[i], r, g, b);
        bytesread += 4;
    }
    break;
default:
    pm_error("Cannot handle %d bit image", bh.bitsperpixel);
    /*NOTREACHED*/
} /* switch on sizes */

if(verbose)
    fprintf(stderr, "Now at position %d\n", bytesread);
```

The header contains the file position where the image data begins. We skip up to that position after checking that we haven't already overshot.

```
/* throw away other header glop */
if(bytesread > bh.pixeloffset)
    pm_error("Corrupt BMP file");

while(bytesread < bh.pixeloffset) {
    (void)pbm_getrawbyte(fd);
    bytesread++;
}
```

Although the image in a BMP file is stored row-by-row, we buffer the entire image in memory because a BMP image, for some reason, is stored from bottom to top rather than from top to bottom, so we have to reverse the order of the rows.

```
image = ppm_allocarray(bh.cols, bh.rows);
if(!image)
    pm_error("Out of memory");
```

Now we loop over the rows. If the file has 24-bit pixels, the pixels are just triples of red, green, and blue data that we can use directly.

```
for(row = bh.rows; --row >= 0; ) {
    pixel *rowdata = image[row];
                    /* pointer to current pixel row */

    if(bh.bitsperpixel == 24) { /* raw pixels */

        for(col = 0; col < bh.cols; col++) {
            int r, g, b;

            r = pbm_getrawbyte(fd);
            g = pbm_getrawbyte(fd);
            b = pbm_getrawbyte(fd);
            bytesread += 3;
            PPM_ASSIGN(rowdata[col], r, g, b);
        } /* cols */
```

For 1-, 4-, and 8-bit pixels, we have to unpack the pixels and look them up in the color map.

```
    } else {
        unsigned bit_store ;
        int bit_count ;
        unsigned int mask = (1 << bh.bitsperpixel) - 1;

        bit_count = 0;
```

```
            for(col = 0; col < bh.cols; col++) {
                int pix;

                if(bit_count <= 0) {     /* need another byte */
                    bit_count = 8;
                    bit_store = pbm_getrawbyte(fd);
                    bytesread++;
                }

                bit_count -= bh.bitsperpixel;
                pix = ( bit_store >> bit_count ) & mask;
                rowdata[col] = colormap[pix];
            } /* cols */
        }
```

Each row is padded to a 4-byte boundary, relative to the beginning of the image data. We skip a few bytes if neccesary to come up to an even boundary.

```
            /* pad input row to even byte */
            while((bytesread-bh.pixeloffset)&3) {
                (void)pbm_getrawbyte(fd);
                bytesread++;
            }

        } /* rows */
```

We have the entire image, so we write it out with the usual library routine and we're done.

```
        /* now write the PPM file, maxval always 255 */
        ppm_writeppm(stdout, image, bh.cols, bh.rows, 255, 0);

        /* that's it */

    } /* readBMP */
```

## 15.3 WRITING BMP FILES

We write BMP files with 1-, 4-, 8-, or 24-bit pixels, using the smallest format that will handle all of the colors in the image. For example, images with more than 16 but no more than 256 colors use 8-bit pixels.

The writer starts conventionally, checking for flags and opening the input file if necessary, and reads in the image. We need to read the entire image for two reasons: to compute the color map and to write out the image from bottom to top in BMP order.

```
/* ppmtobmp.c - read a portable pixmap and produce a BMP file
*/

#include "bmp.h"
#include "ppm.h"
#include "ppmcmap.h"
#include <assert.h>

#define MAXCOLORS 256

void writeBMP ARGS((FILE *, pixel**, int, int, int,
int, colorhist_vector));

static int verbose = FALSE;
static char usage[] = "[-verbose] [ppmfile]";

void
main( argc, argv )
int argc;
char* argv[];
{
    FILE* ifp;
    int argn, rows, cols, colors, bitsperpixel;
    pixval maxval;
    pixel** pixels;
    colorhist_vector chv;

    ppm_init( &argc, argv );

    argn = 1;

    while ( argn < argc && argv[argn][0] == '-'
                        && argv[argn][1] != '\0' )
        {
    if ( pm_keymatch( argv[argn], "-verbose", 2 ) )
        verbose = TRUE;
    else
        pm_usage( usage );
    ++argn;
    }

if ( argn < argc ) {
    ifp = pm_openr( argv[argn] );
    ++argn;
} else
    ifp = stdin;

if ( argn != argc )
    pm_usage( usage );
```

```
    pixels = ppm_readppm( ifp, &cols, &rows, &maxval );

    pm_close( ifp );
```

We compute the color map using the standard library routine and select an appropriate pixel size. Note that if the color map has more than 256 entries, we treat the file as unmappable, since BMP maps are limited to 256 colors.

```
    /* Figure out the colormap. */
    pm_message( "computing colormap..." );
    chv = ppm_computecolorhist( pixels, cols, rows, MAXCOLORS,
          &colors );
    if ( chv == (colorhist_vector) 0 ) {
        if(verbose)
                    pm_message("too many colors - writing
                                        true color BMP file");
        bitsperpixel = 24;
    }
    pm_message( "%d colors found", colors );
    if(colors > 256) {
        if(verbose)
                    pm_message("too many colors - writing
                                        true color BMP file");
        ppm_freecolorhist( chv );
        bitsperpixel = 24;
        chv = 0;
    } else if(colors > 16) {
        bitsperpixel = 8;
        if(verbose)
                    pm_message("writing 8 bit BMP file");
    } else if(colors > 2) {
        bitsperpixel = 4;
        if(verbose)
                    pm_message("writing 4 bit BMP file");
    } else {
        bitsperpixel = 1;
        if(verbose)
                    pm_message("writing 1 bit BMP file");
    }
```

Now that we have the image, the pixel size, and the map, we can call the BMP encoder, after which we're done.

```
    writeBMP(stdout, pixels, bitsperpixel, rows, cols, colors, chv);

    ppm_freecolorhist(chv);
    ppm_freearray(pixels, rows);

} /* main */
```

## Encoding BMP Files

A few macros compute the size of the file header, the size of each line, and the size of the encoded image. We need to precompute these values, since the file header includes size information.

```
#define BMP_HEADERSIZE                          (3 * 2 + 4 * 12)
#define BMP_BYTESPERLINE( width, bits)
        ((((width) * (bits) + 31) / 32) * 4)
#define BMP_PIXELSIZE(width, height, bits) \
    (BMP_BYTESPERLINE(width,bits) * height)
```

The encoding routine starts by computing cmapsize, the size of the color map, and datasize, the size of the image. Armed with those values, we can fill in all of the meaningful fields in the header and write out the header.

```
/*
 * Write BMP file to fd from image in pixels, encode bitsperpixel
 * bits, size nrows by ncols, number of colors given, using
 * color map chv
 */
void
writeBMP(fd, pixels, bitsperpixel, nrows, ncols, colors, chv)
FILE *fd;
pixel** pixels;
int bitsperpixel;
int nrows, ncols;
int colors;
colorhist_vector chv;
{
    colorhash_table cht;
    struct BmpHeader bh;
int datasize, cmapsize;
int byteswritten;
int row, col;

if(bitsperpixel == 24)      /* size of color map */
    cmapsize = 0;
else
    cmapsize = colors * 4;

datasize = BMP_PIXELSIZE(ncols, nrows, bitsperpixel);

/* set up header */

bh.magic1 = 'B';
bh.magic2 = 'M';
bh.filesize = BMP_HEADERSIZE + cmapsize + datasize;
```

```
bh.res1 = bh.res2 = 0;
bh.pixeloffset = BMP_HEADERSIZE + cmapsize;

bh.bmisize = 40;    /* magic number */
bh.cols = ncols;
bh.rows = nrows;
bh.planes = 1;
bh.bitsperpixel = bitsperpixel;
bh.compression = 0;
bh.cmpsize = datasize;  /* size of compressed image */
bh.xscale = 0;      /* pixels per meter */
bh.yscale = 0;
bh.colors = colors;     /* number of colors used */
bh.impcolors = colors;  /* they're all important */
```

We write the header a field at a time to avoid byte-order problems.

```
/* now write the header */
putc(bh.magic1, fd);
putc(bh.magic2, fd);
pm_writelittlelong(fd, bh.filesize);
pm_writelittleshort(fd, bh.res1);
pm_writelittleshort(fd, bh.res2);
pm_writelittlelong(fd, bh.pixeloffset);
pm_writelittlelong(fd, bh.bmisize);
pm_writelittlelong(fd, bh.cols);
pm_writelittlelong(fd, bh.rows);
pm_writelittleshort(fd, bh.planes);
pm_writelittleshort(fd, bh.bitsperpixel);
pm_writelittlelong(fd, bh.compression);
pm_writelittlelong(fd, bh.cmpsize);
pm_writelittlelong(fd, bh.xscale);
pm_writelittlelong(fd, bh.yscale);
pm_writelittlelong(fd, bh.colors);
pm_writelittlelong(fd, bh.impcolors);
```

If the file has a color map, we write it directly from the color histogram, 3 bytes of pixel data and one spare per entry. Then we make a color hash table for quick reference while writing out the image.

```
/* write the color map */
if(cmapsize) {
    int i;

    for(i = 0; i < colors; i++) {
        putc(PPM_GETB(chv[i].color), fd);
        putc(PPM_GETG(chv[i].color), fd);
        putc(PPM_GETR(chv[i].color), fd);
        putc(0, fd);        /* spare byte */
```

```
    }
}
byteswritten = BMP_HEADERSIZE + cmapsize;

/* Make a hash table for fast color lookup. */
if(bitsperpixel < 24)
    cht = ppm_colorhisttocolorhash( chv, colors );
```

Now we write the file a row at a time, bottom to top. If the file is true color, we just write out the components of each pixel.

```
/* write by row, inverting the image */
for(row = nrows; --row >= 0; ) {
    pixel *rowdata = pixels[row];
                        /* current row of PPM pixels */

if(bitsperpixel == 24) {        /* true color */

    for(col = 0; col < ncols; col++) {
        pixel p = rowdata[col];

        putc(PPM_GETR(p), fd);
        putc(PPM_GETG(p), fd);
        putc(PPM_GETB(p), fd);
        byteswritten += 3;
    } /* cols */

} /* true color */
```

If the image is color-mapped, we have to look up each pixel using ppm_lookupcolor( ) and pack the pixels into bytes before writing them. The packing code is quite simple and takes advantage of the fact that integral numbers of pixels always fit in a word.

```
else {  /* color mapped */
    int pixbuf; /* build up pixels here */
    int nbits = 0;  /* bits written so far */

    for(col = 0; col < ncols; col++) {
        int pval = ppm_lookupcolor( cht, &rowdata[col] );

        pixbuf = (pixbuf << bitsperpixel) | pval;

        nbits += bitsperpixel;
        assert(nbits <= 8);
        if(nbits == 8) {
            putc(pixbuf, fd);
            pixbuf = 0;
            nbits = 0;
```

```
                byteswritten++;
            }
        } /* cols */

        if(nbits > 0) { /* write any partially filled byte */
            putc(pixbuf, fd);
            byteswritten++;
        }

    } /* color mapped */
```

Each row is padded out to a multiple of 4 bytes.

```
            while( (bytesread-bh.pixeloffset)&3 ) {
                                    /* pad to long boundary */
                putc(0, fd);
                byteswritten++;
            }
        } /* rows */
```

A quick sanity check verifies that we wrote as much data as we thought we did, and we're done.

```
    if(byteswritten != bh.filesize)
        pm_error("Size is %d, should be %d\n",
byteswritten, bh.filesize);
} /* writeBMP */
```

CHAPTER 16

# Targa

Targa file format, also known as TGA format, was defined by Truevision to support their line of image capture and display cards. Targa files can handle both color-mapped and true color images, but they are most commonly used for true color scanned and ray-traced photographic images. Targa has become the de facto standard for all sorts of full-color bitmap images.

A run-length compression option is available but infrequently used, since scanned photos rarely contain blocks of exactly the same color. (Compression does help on computer-generated images, which often do contain blocks of solid color.) Color images can be 15 or 24 bits per pixel, i.e., 5 or 8 bits per component. Images are sometimes stored as 16 or 32 bits per pixel, with the extra bit(s) being "alpha channel" information used to control how the image is overlaid on others. Eight-bit grayscale images are also reasonably common.

Targa files start with a header that contains the image type and encoding, followed by an optional image description, optional color map, and the image data. Truevision has recently defined an upward-compatible 2.0 image format with a lot of extra information at the end of the file after the image data. Since there are not a lot of 2.0 images yet, and since 2.0 images can be read successfully by a 1.0 reader, we don't consider the 2.0 extensions here. A few features found in older Targa files, notably interleaved image rows, are officially deprecated by the current Truevision standard, but since interleaved images are still around (and it is not hard to handle them), we support them in the reader code.

## 16.1 TARGA INCLUDE FILE

A common include file describes the Targa file header and the type codes that describe all of the supported formats.

```
/* tga.h - header file for Targa files
*/
```

```
/* Header definition. */
struct ImageHeader {
    unsigned char IDLength;
                  /* length of Identifier String */
    unsigned char CoMapType;                  /* 0 = no map */
    unsigned char ImgType;
                  /* image type (see below for values) */
    unsigned char Index_lo, Index_hi;
                  /* index of first color map entry */
    unsigned char Length_lo, Length_hi;
                  /* number of entries in color map */
    unsigned char CoSize;
                  /* size of color map entry (15,16,24,32)*/
    unsigned char X_org_lo, X_org_hi;        /* x origin of image */
    unsigned char Y_org_lo, Y_org_hi;        /* y origin of image */
    unsigned char Width_lo, Width_hi;        /* width of image */
    unsigned char Height_lo, Height_hi;        /* height of image */
    unsigned char PixelSize;          /* pixel size (8,16,24,32) */
    unsigned char AttBits;
                  /* 4 bits, number of attribute bits per pixel */
    unsigned char LRbit;          /* 1 bit, origin: 0=left, 1=right */
    unsigned char OrgBit;
                  /* 1 bit, origin: 0=lower left, 1=upper left */
    unsigned char IntrLve;          /* 2 bits, interleaving flag */
    };

typedef char ImageIDField[256];

/* Definitions for image types. */
#define TGA_Null 0
#define TGA_Map 1          /* uncompressed color mapped */
#define TGA_RGB 2          /* uncompressed true color */
#define TGA_Mono 3        /* uncompressed grayscale */
#define TGA_RLEMap 9        /* RLE compressed color mapped */
#define TGA_RLERGB 10        /* RLE compressed true color */
#define TGA_RLEMono 11        /* RLE compressed grayscale */
#define TGA_CompMap 32        /* obsolete types */
#define TGA_CompMap4 33

/* Definitions for interleave flag. */
#define TGA_IL_None 0
#define TGA_IL_Two 1
#define TGA_IL_Four 2
```

## 16.2 READING TARGA FILES

As usual, the Targa reader starts by checking arguments and opening files.

```
/*   tgatoppm.c - read a Truevision Targa file and write a
 *   portable pixmap
 */
#include "ppm.h"
#include "tga.h"

#ifdef MSDOS
#define MAXCOLORS 256
      /* 16K pixels are more than a 64K segment */
#else
#define MAXCOLORS 16384
#endif

static int mapped, rlencoded;
static pixel ColorMap[MAXCOLORS];

static void readtga ARGS(( FILE* ifp, struct
            ImageHeader* tgaP ));
static void get_map_entry ARGS(( FILE* ifp, pixel*
            Value, int Size ));
static void get_pixel ARGS(( FILE* ifp, pixel* dest,
            int Size ));
static unsigned char getbyte ARGS(( FILE* ifp ));

void
main( argc, argv )
    int argc;
    char* argv[];
    {
    struct ImageHeader tga_head;
    int i;
    unsigned int temp1, temp2;
    FILE* ifp;
    int argn, debug, rows, cols, row, col, realrow,
        truerow, baserow;
    int maxval;
    pixel** pixels;
    char* usage = " [-debug] [tgafile]";

    ppm_init( &argc, argv );

    argn = 1;
    debug = 0;

    if ( argn < argc && argv[argn][0] == '-'
                     && argv[argn][1] != '\0' )
        {
        if ( pm_keymatch( argv[argn], "-debug", 2 ) )
```

```
            debug = 1;
        else
            pm_usage( usage );
        ++argn;
        }

    if ( argn != argc )
        {
        ifp = pm_openr( argv[argn] );
        ++argn;
        }
    else
        ifp = stdin;

    if ( argn != argc )
        pm_usage( usage );
```

We read the image header. (The header reading routine skips over the ID string, if one is present.) If the user requested, we dump out all of the header fields. Targa files store 2-byte quantities in little-endian format. To avoid byte-order problems the code here treats each of the bytes as a separate field and assembles the values on the fly as needed.

```
    /* Read the Targa file header. */
    readtga( ifp, &tga_head );
    if ( debug )
        {
        pm_message( "IDLength = %d\n", (int) tga_head.IDLength );
        pm_message( "CoMapType = %d\n", (int) tga_head.CoMapType );
        pm_message( "ImgType = %d\n", (int) tga_head.ImgType );
        pm_message( "Index_lo = %d\n", (int) tga_head.Index_lo );
        pm_message( "Index_hi = %d\n", (int) tga_head.Index_hi );
        pm_message( "Length_lo = %d\n", (int) tga_head.Length_lo );
        pm_message( "Length_hi = %d\n", (int) tga_head.Length_hi );
        pm_message( "CoSize = %d\n", (int) tga_head.CoSize );
        pm_message( "X_org_lo = %d\n", (int) tga_head.X_org_lo );
        pm_message( "X_org_hi = %d\n", (int) tga_head.X_org_hi );
        pm_message( "Y_org_lo = %d\n", (int) tga_head.Y_org_lo );
        pm_message( "Y_org_hi = %d\n", (int) tga_head.Y_org_hi );
        pm_message( "Width_lo = %d\n", (int) tga_head.Width_lo );
        pm_message( "Width_hi = %d\n", (int) tga_head.Width_hi );
        pm_message( "Height_lo = %d\n", (int) tga_head.Height_lo );
        pm_message( "Height_hi = %d\n", (int) tga_head.Height_hi );
        pm_message( "PixelSize = %d\n", (int) tga_head.PixelSize );
        pm_message( "AttBits = %d\n", (int) tga_head.AttBits );
        pm_message( "Rsrvd = %d\n", (int) tga_head.Rsrvd );
        pm_message( "OrgBit = %d\n", (int) tga_head.OrgBit );
        pm_message( "IntrLve = %d\n", (int) tga_head.IntrLve );
        }
```

We assemble the image size and check that the format is one that we understand.

```
rows = ( (int) tga_head.Height_lo ) + ( (int)
               tga_head.Height_hi ) * 256;
cols = ( (int) tga_head.Width_lo ) + ( (int)
               tga_head.Width_hi ) * 256;

switch ( tga_head.ImgType )
    {
    case TGA_Map:
    case TGA_RGB:
    case TGA_Mono:
    case TGA_RLEMap:
    case TGA_RLERGB:
    case TGA_RLEMono:
    break;

    default:
    pm_error( "unknown Targa image type %d",
              tga_head.ImgType );
    }
```

If there's a color map, be sure that it's in format 1, the only one we understand (and the only one in common use). We also check that we can handle the number of bits per component in the color map.

```
if ( tga_head.ImgType == TGA_Map ||
     tga_head.ImgType == TGA_RLEMap )
    { /* Color-mapped image */
    if ( tga_head.CoMapType != 1 )
        pm_error(
            "mapped image (type %d) with color map type
             != 1",
            tga_head.ImgType );
mapped = 1;
/* Figure maxval from CoSize. */
switch ( tga_head.CoSize )
    {
    case 8:
    case 24:
    case 32:
    maxval = 255;
    break;

    case 15:
    case 16:
    maxval = 31;
    break;
```

```
            default:
            pm_error(
                "unknown colormap pixel size - %d",
                 tga_head.CoSize );
            }
        if ( maxval > PPM_MAXMAXVAL )
            pm_error(
"CoSize is too large - try reconfiguring with PGM_BIGGRAYS\n"
"    or without PPM_PACKCOLORS" );
        }
    else
```

If the image is not color-mapped, we again check that the pixels are a size we can handle.

```
    { /* No colormap, so figure maxval from PixelSize. */
    mapped = 0;
    switch ( tga_head.PixelSize )
        {
        case 8:
        case 24:
        case 32:
        maxval = 255;
        break;

        case 15:
        case 16:
        maxval = 31;
        break;

        default:
        pm_error( "unknown pixel size - %d",
        tga_head.PixelSize );
        }
    if ( maxval > PPM_MAXMAXVAL )
        pm_error(
"PixelSize is too large - try reconfiguring with PGM_BIGGRAYS\n"
"    or without PPM_PACKCOLORS" );
        }
```

Now we read in the color map, if there is one. (True-color and grayscale files may have a color map, even though the map isn't needed to interpret the image.) The color map need not contain a full set of entries. If, for example, pixels are 8 bits, a full-color map is 256 entries, but the color map in the file might only contain entries 100 to 200. This sort of limited range often occurs when images with independent color maps need to be combined into a larger image with a combined color map.

We also check whether we need to handle run-length encoding.

```
/* If required, read the color map information. */
if ( tga_head.CoMapType != 0 )
    {
    temp1 = tga_head.Index_lo + tga_head.Index_hi * 256;
    temp2 = tga_head.Length_lo + tga_head.Length_hi * 256;
    if ( ( temp1 + temp2 + 1 ) >= MAXCOLORS )
        pm_error( "too many colors - %d",
        ( temp1 + temp2 + 1 ) );
    for ( i = temp1; i < ( temp1 + temp2 ); ++i )
        get_map_entry( ifp, &ColorMap[i], (int)
        tga_head.CoSize );
    }
```

```
/* Check run-length encoding. */
if ( tga_head.ImgType == TGA_RLEMap ||
     tga_head.ImgType == TGA_RLERGB ||
     tga_head.ImgType == TGA_RLEMono )
    rlencoded = 1;
else
    rlencoded = 0;
```

We have the image size, pixel size and encoding, and row order, so we are ready to read in the image. The main complication here comes from the many possible row orders in the file. Depending on the origin bit tga_head.OrgBit, the rows may run from top to bottom or from bottom to top. Also, if the deprecated interleave field tga_head.IntrLve is non-zero, the image is two-way or four-way interleaved. If the image is four-way interleaved, the image first contains rows 0, 4, 8, 12, . . . , then 1, 5, 9, . . . , then 2, 6, 10, . . . , and then 3, 7, 11, . . . . The variable row counts rows in the image file, truerow is the row number accounting for interleave, and realrow is the row number, taking into account both the interleave and the top or bottom origin. (The last two variables could have been assigned better names.) Once we've figured out which row this is, we scan across the row calling get_pixel( ) to get the actual pixel values. We don't handle right-to-left pixel order, since no images use it, but if necessary we could easily do so by adding an alternate for-loop that scanned from higher-numbered to lower-numbered columns.

```
/* Read the Targa file body and convert to portable format. */
pixels = ppm_allocarray( cols, rows );
truerow = 0;
baserow = 0;
for ( row = 0; row < rows; ++row )
    {
    realrow = truerow;

    /* account for top or bottom origin */
    if ( tga_head.OrgBit == 0 )
        realrow = rows - realrow - 1;
```

```
for ( col = 0; col < cols; ++col )
    get_pixel( ifp, &(pixels[realrow][col]),
               (int) tga_head.PixelSize );

/* compute next row allowing for interleave */
if ( tga_head.IntrLve == TGA_IL_Four )
    truerow += 4;
else if ( tga_head.IntrLve == TGA_IL_Two )
    truerow += 2;
else
    ++truerow;

/* ran off the bottom, start next scan pass */
if ( truerow >= rows )
    truerow = ++baserow;
}
```

Finally, we have the whole image, so we write it out and we're done.

```
pm_close( ifp );

ppm_writeppm( stdout, pixels, cols, rows, (pixval) maxval, 0 );
pm_close( stdout );

exit( 0 );
}
```

## Auxiliary Routines for Reading Targa Files

The routine readtga( ) reads the Targa header a byte at a time. If the IDLength is non-zero, we read and discard the ID string.

```
/* read Targa header from file ifp into ImageHeader *tgaP */
static void
readtga( ifp, tgaP )
    FILE* ifp;
    struct ImageHeader* tgaP;
    {
    unsigned char flags;
    ImageIDField junk;

    tgaP->IDLength = getbyte( ifp );
    tgaP->CoMapType = getbyte( ifp );
    tgaP->ImgType = getbyte( ifp );
    tgaP->Index_lo = getbyte( ifp );
    tgaP->Index_hi = getbyte( ifp );
    tgaP->Length_lo = getbyte( ifp );
    tgaP->Length_hi = getbyte( ifp );
    tgaP->CoSize = getbyte( ifp );
```

```
    tgaP->X_org_lo = getbyte( ifp );
    tgaP->X_org_hi = getbyte( ifp );
    tgaP->Y_org_lo = getbyte( ifp );
    tgaP->Y_org_hi = getbyte( ifp );
    tgaP->Width_lo = getbyte( ifp );
    tgaP->Width_hi = getbyte( ifp );
    tgaP->Height_lo = getbyte( ifp );
    tgaP->Height_hi = getbyte( ifp );
    tgaP->PixelSize = getbyte( ifp );
    flags = getbyte( ifp );
    tgaP->AttBits = flags & 0xf;
    tgaP->Rsrvd = ( flags & 0x10 ) >> 4;
    tgaP->OrgBit = ( flags & 0x20 ) >> 5;
    tgaP->IntrLve = ( flags & 0xc0 ) >> 6;

    if ( tgaP->IDLength != 0 )
        fread( junk, 1, (int) tgaP->IDLength, ifp );
} /* readtga */
```

Each color-map entry is read by get_map_entry( ). Depending on the color-map entry size, it reads 1, 2, or 3 bytes and breaks it up into the three color components. Each entry is an integral number of bytes, so the 15-bit format is handled in the same way as the 16-bit format. (The high bit of the 16-bit format is an alpha value, but we ignore that.)

```
/* read a color map entry of Size bits from file ifp into PPM pixel
 * Value */
static void
get_map_entry( ifp, Value, Size )
    FILE* ifp;
    pixel* Value;
    int Size;
    {
    unsigned char j, k, r, g, b;

    /* Read appropriate number of bytes, break into
     *   rgb & put in map.
     */
    switch ( Size )
        {
        case 8:     /* Grayscale, read and triplicate. */
        r = g = b = getbyte( ifp );
        break;

        case 16:    /* 5 bits each of red green and blue. */
        case 15:          /* Watch for byte order. */
        j = getbyte( ifp );
        k = getbyte( ifp );
        r = ( k & 0x7C ) >> 2;
```

```
    g = ( ( k & 0x03 ) << 3 ) + ( ( j & 0xE0 ) >> 5 );
    b = j & 0x1F;
    break;

    case 32:
    case 24:   /* 8 bits each of blue green and red. */
    b = getbyte( ifp );
    g = getbyte( ifp );
    r = getbyte( ifp );
    if ( Size == 32 )
        (void) getbyte( ifp );
                    /* Read alpha byte & throw away. */
    break;

    default:
        pm_error( "unknown colormap pixel size (#2) - %d",
        Size );
        }
    PPM_ASSIGN( *Value, r, g, b );
    } /* get_map_entry */
```

Each pixel is read by get_pixel( ). It has the same multiple pixel-size code as get_map_entry( ) and has the extra complication of RLE compression. RLE-compressed files use a scheme similar to PackBits. Each group of pixels in the file is preceded by a control byte in which the low seven bits are a repeat count (zero-based, so a value of 0 means a repeat count of 1), and the high bit indicates a repeated pixel if set or a group of literal pixels if clear.

Within get_pixel( ), the static variable RLE_count tracks how many of the current group remain to be processed, and RLE_flag indicates whether it's a repeating group or not. If it is a repeating group, the static variables Red, Grn, and Blu keep the components of the repeated pixel in a true-color image, or the variable l keeps the color index in a color-mapped image.

```
    /* read the next pixel of Size bits from file ifp into PPM
     * pixel *dest
     */

    static int RLE_count = 0, RLE_flag = 0;

    static void
    get_pixel( ifp, dest, Size )
        FILE* ifp;
        pixel* dest;
        int Size;
        {
        static pixval Red, Grn, Blu;
        unsigned char j, k;
        static unsigned int l;
```

```
/* Check if run length encoded. */
if ( rlencoded )
    {
    if ( RLE_count == 0 )
        {       /* Need to get the next group */
        unsigned char i;
        i = getbyte( ifp );
        RLE_flag = ( i & 0x80 );
        if ( RLE_flag == 0 )

                /* Stream of unencoded pixels. */
                   RLE_count = i + 1;
        else
                /* Single pixel replicated. */
                   RLE_count = i - 127;
                /* Decrement count & get pixel. */
            --RLE_count;
            }
        else
            { /* Have already read count & (at least)
               * first pixel. */
            --RLE_count;
            if ( RLE_flag != 0 )
                    /* Replicated pixels, use existing values. */
                       goto PixEncode;
            }
        }
    /* Read another pixel,
         * read appropriate number of bytes, break into RGB.
     */
    switch ( Size )
    {
    case 8:         /* Grayscale, read and triplicate. */
    Red = Grn = Blu = l = getbyte( ifp );
    break;

    case 16:        /* 5 bits each of red green and blue. */
    case 15:        /* Watch byte order. */
    j = getbyte( ifp );
    k = getbyte( ifp );
    l = ( (unsigned int) k << 8 ) + j;
    Red = ( k & 0x7C ) >> 2;
    Grn = ( ( k & 0x03 ) << 3 ) + ( ( j & 0xE0 ) >> 5 );
    Blu = j & 0x1F;
    break;

    case 32:
```

```
case 24:        /* 8 bits each of blue green and red. */
Blu = getbyte( ifp );
Grn = getbyte( ifp );
Red = getbyte( ifp );
if ( Size == 32 )
    (void) getbyte( ifp );
                /* Read alpha byte & throw away. */
l = 0;
break;

default:
pm_error( "unknown pixel size (#2) - %d", Size );
}

PixEncode:
    if ( mapped )
                *dest = ColorMap[l];
                      /* get value from color map */
    else
                PPM_ASSIGN( *dest, Red, Grn, Blu );
    }
```

Finally, yet another getbyte( ) routine reads bytes and checks for errors. It would have been appropriate to use the library routine ppm_getrawbyte( ) instead.

```
static unsigned char
getbyte( ifp )
    FILE* ifp;
    {
    int c;

    c = getc(ifp);
    if(c == EOF)
        pm_error( "EOF / read error" );

return c;
    }
```

## 16.3 WRITING TARGA FILES

Writing a Targa file is simpler than reading one because there are far fewer situations to handle. We only handle 24-bit color and grayscale images, with or without color mapping. The user can specify an encoding on the command line, or more commonly the program figures out an appropriate encoding depending on the type of image.

Our Targa writer starts with the usual switch and file name handling code. The –name argument lets the user specify an image name, or it picks a default name based on the input file name. The intent was probably to store the name as the

ID field in the file, but in fact this code always writes a null ID. That turns out to be a reasonable thing to do, since so few Targa files actually contain ID fields that many Targa readers don't work on files with IDs.

```
/* ppmtotga.c - read a portable pixmap and produce a
 *   Truevision Targa file
 */

#include "ppm.h"
#include "ppmcmap.h"
#include "tga.h"

/* Max number of colors allowed for colormapped output. */
#define MAXCOLORS 256

/* Forward routines. */
static void writetga ARGS(( struct ImageHeader* tgaP,
            char* id ));
static void put_map_entry ARGS(( pixel* valueP,
            int size, pixval maxval ));
static void compute_runlengths ARGS(( int cols, pixel*
            pixelrow, int* runlength ));
static void put_pixel ARGS(( pixel* pP, int imgtype,
            pixval maxval, colorhash_table cht ));
static void put_mono ARGS(( pixel* pP, pixval maxval ));
static void put_map ARGS(( pixel* pP, colorhash_table cht ));
static void put_rgb ARGS(( pixel* pP, pixval maxval ));

/* Routines. */

void
main( argc, argv )
    int argc;
char* argv[];
{
FILE* ifp;
pixel** pixels;
register pixel* pP;
pixel p;
int argn, rle_flag, rows, cols, ncolors, row, col, i,
format, realrow;
pixval maxval;
colorhist_vector chv;
colorhash_table cht;
char out_name[100];
char* cp;
int* runlength;
char* usage = "[-name <tganame>] [-mono|-cmap|-rgb]
              [-norle] [ppmfile]";
```

```
    struct ImageHeader tgaHeader;

    ppm_init( &argc, argv );
    out_name[0] = '\0';

    /* Check for command line options. */
    argn = 1;
    tgaHeader.ImgType = TGA_Null;
                    /* default is we choose encoding */
    rle_flag = 1;
                    /* default to RLE compression */

    while ( argn < argc && argv[argn][0] == '-'
                        && argv[argn][1] != '\0' )
        {
    if ( pm_keymatch( argv[argn], "-name", 2 ) )
        {
        ++argn;
        if ( argn == argc )
            pm_usage( usage );
        (void) strcpy( out_name, argv[argn] );
        }
    else if ( pm_keymatch( argv[argn], "-cmap", 2 ) )
        tgaHeader.ImgType = TGA_Map;
    else if ( pm_keymatch( argv[argn], "-mono", 2 ) )
        tgaHeader.ImgType = TGA_Mono;
    else if ( pm_keymatch( argv[argn], "-rgb", 2 ) )
        tgaHeader.ImgType = TGA_RGB;
    else if ( pm_keymatch( argv[argn], "-norle", 2 ) )
        rle_flag = 0;
    else
        pm_usage( usage );
    ++argn;
    }

    if ( argn != argc )
        {
        /* Open the input file. */
        ifp = pm_openr( argv[argn] );

        /* If output filename not specified, use input filename
                * minus any suffix after a dot. */
        if ( out_name[0] == '\0' )
            {
            (void) strcpy( out_name, argv[argn] );
            cp = index( out_name, '.' );
            if ( cp != 0 )
                *cp = '\0';          /* remove extension */
            if ( strcmp( out_name, "-" ) == 0 )
```

```
                (void) strcpy( out_name, "noname" );
            }

        ++argn;
        }
    else
        {
        /* No input file specified. */
        ifp = stdin;
        if ( out_name[0] == '\0' )
            (void) strcpy( out_name, "noname" );
        }

    if ( argn != argc )
        pm_usage( usage );
```

We have to read in the whole file here because we may have to compute a color map.

```
/* Read in the ppm file. */
ppm_readppminit( ifp, &cols, &rows, &maxval, &format);
pixels = ppm_allocarray( cols, rows );
for ( row = 0; row < rows; ++row )
    ppm_readppmrow( ifp, pixels[row], cols, maxval, format );
pm_close( ifp );
```

We use PPM_FORMAT_TYPE( ) to identify the actual type of input file. (Recall that the PPM routines can also read PGM and PBM files.)

PGM and PBM files turn into grayscale mono images, and PPM files turn into color-mapped files if there are no more than MAXCOLORS (256) colors; otherwise they turn into true color. The user can force a larger format, e.g., true color for a PGM input.

```
/* Figure out the colormap. */
switch ( PPM_FORMAT_TYPE( format ) )
    {
    case PPM_TYPE:
    if ( tgaHeader.ImgType == TGA_Mono )
        pm_error( "input is not a graymap,
                filter through ppmtopgm first" );
    if ( tgaHeader.ImgType == TGA_Null ||
         tgaHeader.ImgType == TGA_Map )
        {
        pm_message( "computing colormap..." );
        chv = ppm_computecolorhist(
            pixels, cols, rows, MAXCOLORS, &ncolors );
        if ( chv == (colorhist_vector) 0 )
            {
```

```
            if ( tgaHeader.ImgType == TGA_Map )
                pm_error(
                    "too many colors - try doing a
                     'ppmquant %d'",
                    MAXCOLORS );
            else
                tgaHeader.ImgType = TGA_RGB;
            }
        else
            {
            pm_message( "%d colors found", ncolors );
            if ( tgaHeader.ImgType == TGA_Null )
                tgaHeader.ImgType = TGA_Map;
            }
        }
    break;

    case PGM_TYPE:
    case PBM_TYPE:
    if ( tgaHeader.ImgType == TGA_Null )
        tgaHeader.ImgType = TGA_Mono;
    else if ( tgaHeader.ImgType == TGA_Map )
        {
        pm_message( "computing colormap..." );
        chv = ppm_computecolorhist(
            pixels, cols, rows, MAXCOLORS, &ncolors );
        if ( chv == (colorhist_vector) 0 )
            pm_error( "can't happen" );
        pm_message( "%d colors found", ncolors );
        }
    break;

    default:
    pm_error( "can't happen" );
    }
```

We use RLE compression unless the user said not to. The array runlength[ ] will keep some auxiliary information about the position in the row where each RLE run begins.

```
if ( rle_flag )
    {
    switch ( tgaHeader.ImgType )
        {
        case TGA_Mono:
        tgaHeader.ImgType = TGA_RLEMono;
        break;
        case TGA_Map:
        tgaHeader.ImgType = TGA_RLEMap;
```

```
            break;
            case TGA_RGB:
            tgaHeader.ImgType = TGA_RLERGB;
            break;
            default:
            pm_error( "can't happen" );
            }
        runlength = (int*) pm_allocrow( cols, sizeof(int) );
        }
```

Now we fill in the header fields. If there is a color map, we make a color hash table for fast access.

```
        tgaHeader.IDLength = 0;
        tgaHeader.Index_lo = 0;
        tgaHeader.Index_hi = 0;
        if ( tgaHeader.ImgType == TGA_Map ||
             tgaHeader.ImgType == TGA_RLEMap )
            {
            /* Make a hash table for fast color lookup. */
            cht = ppm_colorhisttocolorhash( chv, ncolors );

            tgaHeader.CoMapType = 1;
            tgaHeader.Length_lo = ncolors % 256;
            tgaHeader.Length_hi = ncolors / 256;
            tgaHeader.CoSize = 24;
            }
        else
            {
            tgaHeader.CoMapType = 0;
            tgaHeader.Length_lo = 0;
            tgaHeader.Length_hi = 0;
            tgaHeader.CoSize = 0;
            }
        if ( tgaHeader.ImgType == TGA_RGB ||
             tgaHeader.ImgType == TGA_RLERGB )
            tgaHeader.PixelSize = 24;
        else
            tgaHeader.PixelSize = 8;
        tgaHeader.X_org_lo = tgaHeader.X_org_hi = 0;
        tgaHeader.Y_org_lo = tgaHeader.Y_org_hi = 0;
        tgaHeader.Width_lo = cols % 256;
        tgaHeader.Width_hi = cols / 256;
        tgaHeader.Height_lo = rows % 256;
        tgaHeader.Height_hi = rows / 256;
        tgaHeader.AttBits = 0;
        tgaHeader.Rsrvd = 0;
        tgaHeader.IntrLve = 0;
```

```
tgaHeader.OrgBit = 0;
```

Now we write the header and, if appropriate, the color map.

```
/* Write out the Targa header. */
writetga( &tgaHeader, (char*) 0 );

if ( tgaHeader.ImgType == TGA_Map ||
                          tgaHeader.ImgType == TGA_RLEMap )
    {
    /* Write out the Targa colormap. */
    for ( i = 0; i < ncolors; ++i )
        put_map_entry( &chv[i].color,
                       tgaHeader.CoSize, maxval );
    }
```

We write the data a row at a time. There is code here to support either top or bottom row origins, although in this code there's no way to set OrgBit.

```
/* Write out the pixels */
for ( row = 0; row < rows; ++row )
    {
    realrow = row;
    if ( tgaHeader.OrgBit == 0 )
        realrow = rows - realrow - 1;
```

For RLE images, we call compute_runlength( ) to fill in the auxiliary array runlength[ ]. Each positive entry in runlength[ ] indicates a repeated pixel, and each negative entry indicates a literal group. Using that information, it is easy to write out the row, using put_pixel( ) to write the actual pixel bits.

```
    if ( rle_flag )
    {
    compute_runlengths( cols, pixels[realrow], runlength );
    for ( col = 0; col < cols; )
        {
        if ( runlength[col] > 0 )
            {
            putchar( 0x80 + runlength[col] - 1 );
            put_pixel(
                &(pixels[realrow][col]),
                tgaHeader.ImgType, maxval, cht );
            col += runlength[col];
            }
        else if ( runlength[col] < 0 )
            {
            putchar( -runlength[col] - 1 );
            for ( i = 0; i < -runlength[col]; ++i )
```

```
                        put_pixel(
                            &(pixels[realrow][col + i]),
                            tgaHeader.ImgType, maxval, cht );
                    col += -runlength[col];
                    }
                else
                    pm_error( "can't happen" );
                }
            }
        else
            {       /* no RLE, just write the pixels */
            for ( col = 0, pP = pixels[realrow]; col < cols;
            ++col, ++pP )
                put_pixel( pP, tgaHeader.ImgType, maxval, cht );
            }
        }
```

Having written all the rows, we're done.

```
    exit( 0 );
    } /* main */
```

## Auxiliary Routines for Writing Targa Files

We use writetga( ) to write the header 1 byte at a time, thereby avoiding byte-order problems.

```
/* write header from *tgaP, ID string from *id */
static void
writetga( tgaP, id )
    struct ImageHeader* tgaP;
    char* id;
    {
    unsigned char flags;

    putchar( tgaP->IDLength );
    putchar( tgaP->CoMapType );
    putchar( tgaP->ImgType );
    putchar( tgaP->Index_lo );
    putchar( tgaP->Index_hi );
    putchar( tgaP->Length_lo );
    putchar( tgaP->Length_hi );
    putchar( tgaP->CoSize );
    putchar( tgaP->X_org_lo );
    putchar( tgaP->X_org_hi );
    putchar( tgaP->Y_org_lo );
    putchar( tgaP->Y_org_hi );
    putchar( tgaP->Width_lo );
    putchar( tgaP->Width_hi );
```

```
    putchar( tgaP->Height_lo );
    putchar( tgaP->Height_hi );
    putchar( tgaP->PixelSize );
    flags = ( tgaP->AttBits & 0xf ) |
            ( ( tgaP->Rsrvd & 0x1 ) << 4 ) |
            ( ( tgaP->OrgBit & 0x1 ) << 5 ) |
            ( ( tgaP->OrgBit & 0x3 ) << 6 );
    putchar( flags );
    if ( tgaP->IDLength )          /* ID string */
        fwrite( id, 1, (int) tgaP->IDLength, stdout );
    }
```

We write each color-map entry with put_map_entry( ). The only complication is the three file formats. Note that for the color formats, we scale the values with PPM_DEPTH( ) to get the maximum color range that the encoding supports.

```
/* write a color map entry from PPM pixel *valueP,
static void
put_map_entry( pixel* valueP, int size, pixval maxval )
    {
    int j;
    pixel p;

    switch ( size )
        {
        case 8:    /* Grayscale. */
        put_mono( valueP, maxval );
        break;

        case 16:  /* 5 bits each of red green and blue. */
        case 15:  /* Watch for byte order. */
                  /* scale components to range 0..31 */
        PPM_DEPTH( p, *valueP, maxval, 31 );
        j = (int) PPM_GETB( p ) | ( (int) PPM_GETG( p ) << 5 ) |
            ( (int) PPM_GETR( p ) << 10 );
        putchar( j % 256 );
        putchar( j / 256 );
        break;

        case 32:
        case 24:  /* 8 bits each of blue green and red. */
        put_rgb( valueP, maxval );
        break;

        default:
        pm_error( "unknown colormap pixel size (#2) - %d", size );
        }
    } /* put_map_entry */
```

RLE encoding depends on the routine compute_runlengths( ), which scans through a row of pixels looking for groups of adjacent identical pixels. It then turns adjacent groups of length 1 into literal groups marked by negative entries in runlength[ ].

```
static void
compute_runlengths( cols, pixelrow, runlength )
    int cols;
    pixel* pixelrow;
    int* runlength;
    {
    int col, start;

    /* Initialize all run lengths to 0.
     *  (This is just an error check.)
     */
    for ( col = 0; col < cols; ++col )
        runlength[col] = 0;

    /* Find runs of identical pixels. */
    for ( col = 0; col < cols; )
        {
        start = col;
        do {
            ++col;
            }
        while ( col < cols &&
                col - start < 128 &&
                PPM_EQUAL( pixelrow[col], pixelrow[start] ) );
        runlength[start] = col - start;
        }

    /* Now look for runs of length-1 runs, and turn them
     * into negative runs.
     */
    for ( col = 0; col < cols; )
        {
        if ( runlength[col] == 1 )
            {
            start = col;
            while ( col < cols &&
                    col - start < 128 &&
                    runlength[col] == 1 )
                {
                runlength[col] = 0;
                ++col;
                }
            runlength[start] = - ( col - start );
            }
```

```
        else
            col += runlength[col];
        }
    }
```

We write each pixel with put_pixel( ). The three different pixel formats (we've already taken care of RLE by this point) are passed to three different routines, shown below.

```
/* write a pixel from PPM pixel *pP, max component value maxval,
 * optionally using color map in hash table cht
 */
static void
put_pixel( pixel* pP, int imgtype, pixval maxval,
           colorhash_table cht )
    {
    switch ( imgtype )
        {
        case TGA_Mono:
        case TGA_RLEMono:
        put_mono( pP, maxval );
        break;
        case TGA_Map:
        case TGA_RLEMap:
        put_map( pP, cht );
        break;
        case TGA_RGB:
        case TGA_RLERGB:
        put_rgb( pP, maxval );
        break;
        default:
        pm_error( "can't happen" );
        }
    }
```

Grayscale pixels are written by put_mono( ), which merely rescales them to the range 0..255 and writes out a byte with the scaled value.

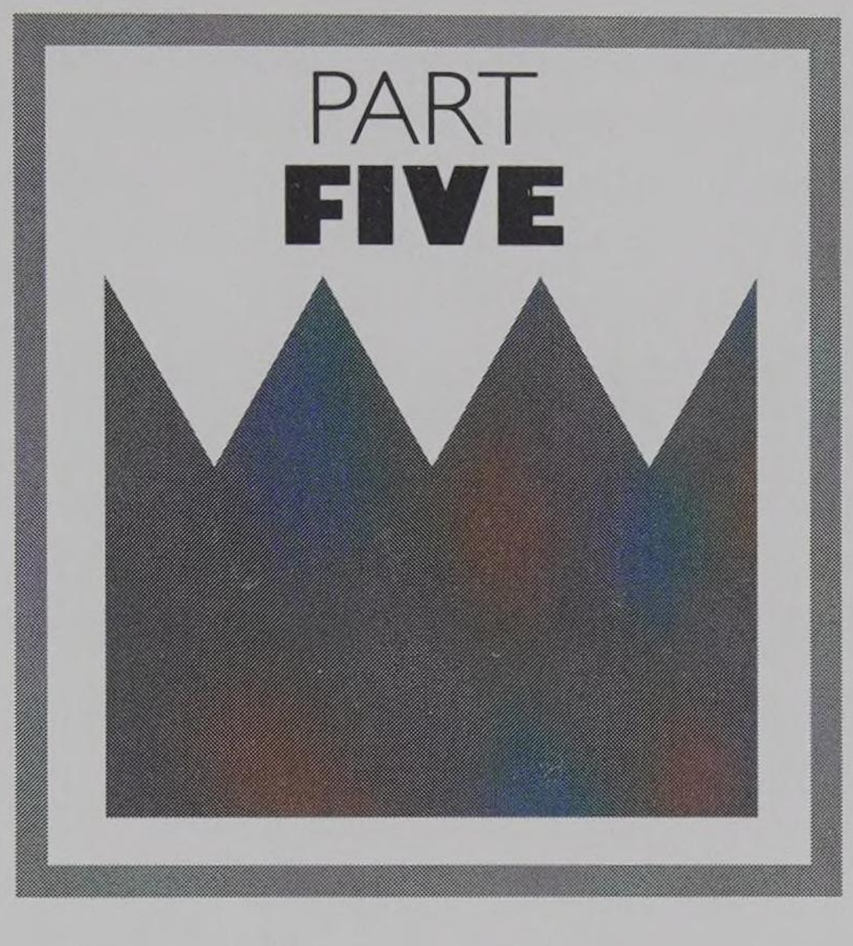
```
static void
put_mono( pixel* pP, pixval maxval )
    {
    PPM_DEPTH( *pP, *pP, maxval, (pixval) 255 );
    putchar( PPM_GETR( *pP ) );
    }
```

Color-mapped pixels are written by put_map( ), which has only to look up the pixel index from the color hash table.

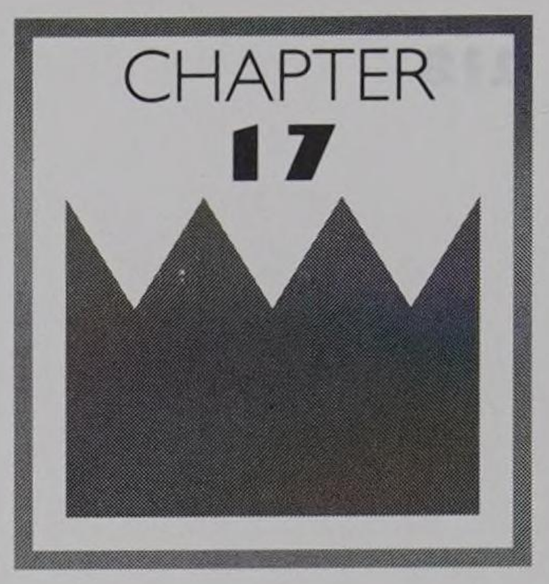
```
static void
put_map( pP, cht )
    pixel* pP;
    colorhash_table cht;
    {
    putchar( ppm_lookupcolor( cht, pP ) );
    }
```

Finally, literal RGB pixels are written by put_rgb( ), which rescales the pixel and writes out the three 8-bit components.

```
static void
put_rgb( pixel* pP, pixval maxval )
    {
    PPM_DEPTH( *pP, *pP, maxval, (pixval) 255 );
    putchar( PPM_GETB( *pP ) );
    putchar( PPM_GETG( *pP ) );
    putchar( PPM_GETR( *pP ) );
    }
```




# Dictionary Compressed Formats




# GIF

Compuserve defined GIF format in 1987 to provide a standard way of encoding the large number of scanned and computer-generated pictures that were becoming popular. It has become one of the most widely used formats for storing color bitmap images because it is well documented, well standardized with few variants, and relatively simple. GIF is intended both as a file format and as a data-stream format for pictures transmitted from host systems to a graphics terminal. It has a few features that are primarily useful in the latter context, such as the ability to include multiple images in sequence, hints on how long an image should be displayed, how it should be overlaid on previous images on the screen, and so forth. These features are encapsulated in *extension blocks,* which we look at only in enough detail to know how to skip over them; the details can be found in the widely available Compuserve GIF spec. Images can be interlaced, with the data stored in the order first every eighth image row, then every fourth row (not counting the rows already sent), then every second row (again not counting the rows already sent), and finally the remaining odd-numbered rows. This lets a graphics terminal that is receiving an image over a slow connection display successively better approximations as the image is received. In files, interlacing is mostly an inconvenience, since the image can usually be displayed in its entirety almost as fast as the first interlace pass can, but we support it anyway since many GIF files are in fact interlaced.

GIF is essentially a straightforward color-mapped format. The most unusual feature of GIF files is that the pixels are compressed using LZW dictionary compression. LZW compression is extremely effective on images with repeating pixel patterns, including solid color areas that have blocks of the same pixel value. It is less effective on scanned photographs, but it's still better than nothing, so that GIF files are usually the most compact popular bitmap format other than the much more complex JPEG. (See Chapter 19.)

Files can contain up to 256 distinct colors, requiring 8 bits per pixel. If the file uses fewer colors, pixels are stored using the minimum number of bits needed. For example, a 64-color file uses 6-bit pixels, and a 2-color bitmap uses single-bit pixels.

A GIF file starts with a header and an optional global color map that is potentially used for all images in the file. Following that are one or more images, each consisting of an image descriptor block, an optional local color map (which is required if there is no global map and overrides the global one if both are present), optional extension blocks, and the compressed image data. In practice, most GIF files contain a global color map and a single image with no extensions.

## 17.1 WRITING GIF FILES

We start by looking at writing a GIF file, because understanding the LZW encoder used in writing GIF files is a prerequisite to understanding the decoder.

Our encoder starts conventionally, checking flags and opening files. The only flag we handle is –interlace, to produce an interlaced GIF file.

```
/* ppmtogif.c - read a portable pixmap and produce a GIF file
*/

#include "ppm.h"
#include "ppmcmap.h"

#define MAXCOLORS 256

/*
 * Typedef for pointer to function returning an int
 */
typedef int (* ifunptr)();

/*
 * a code_int must be able to hold 2**BITS values of type int,
 * and also -1
 */
typedef short int       code_int; /* was int */
typedef long int        count_int;

static int colorstobpp ARGS(( int colors ));
static int GetPixel ARGS(( int x, int y ));
static void BumpPixel ARGS(( void ));
static int GIFNextPixel ARGS(( ifunptr getpixel ));
static void GIFEncode ARGS(( FILE* fp, int GWidth,
            int GHeight, int GInterlace, int Background,
            int BitsPerPixel, int Red[], int Green[],
            int Blue[], ifunptr GetPixel ));
static void Putword ARGS(( int w, FILE* fp ));
static void compress ARGS(( int init_bits, FILE* outfile,
```

```
                ifunptr ReadValue ));
static void output ARGS(( code_int code ));
static void cl_block ARGS(( void ));
static void cl_hash ARGS(( count_int hsize ));
static void writeerr ARGS(( void ));
static void char_init ARGS(( void ));
static void char_out ARGS(( int c ));
static void flush_char ARGS(( void ));

static pixel** pixels;
static colorhash_table cht;

void
main( argc, argv )
    int argc;
char* argv[];
{
FILE* ifp;
int argn, rows, cols, colors, i, BitsPerPixel;
int interlace;
pixval maxval;
colorhist_vector chv;
int Red[MAXCOLORS], Green[MAXCOLORS], Blue[MAXCOLORS];
char* usage = "[-interlace] [ppmfile]";

ppm_init( &argc, argv );

argn = 1;
interlace = 0;

while ( argn < argc && argv[argn][0] == '-'
                    && argv[argn][1] != '\0' )
    {
    if ( pm_keymatch( argv[argn], "-interlace", 2 ) )
        interlace = 1;
    else if ( pm_keymatch( argv[argn], "-nointerlace", 2 ) )
        interlace = 0;
    else
        pm_usage( usage );
    ++argn;
    }

if ( argn < argc )
{
ifp = pm_openr( argv[argn] );
++argn;
}
else
ifp = stdin;
```

```
if ( argn != argc )
pm_usage( usage );
```

Now we read in the image and compute the color map. The color map in a GIF file is stored with 8 bits per component, so we rescale the colors in the map if need be. Then we set BitsPerPixel to be the minimum number of bits needed to represent the number of colors present in the image, make the usual color hash table to speed later color lookups, and call GIFEncode( ), the routine that actually writes a GIF file.

```
pixels = ppm_readppm( ifp, &cols, &rows, &maxval );

pm_close( ifp );

/* Figure out the colormap. */
pm_message( "computing colormap..." );
chv = ppm_computecolorhist( pixels, cols, rows,
      MAXCOLORS, &colors );
if ( chv == (colorhist_vector) 0 )
pm_error(
    "too many colors - try doing a 'ppmquant %d'",
     MAXCOLORS );
pm_message( "%d colors found", colors );

/* Now turn the ppm colormap into the appropriate GIF colormap. */
if ( maxval > 255 )
pm_message(
    "maxval is not 255 - automatically rescaling colors" );
for ( i = 0; i < colors; ++i )
{
if ( maxval == 255 )
    {
    Red[i] = PPM_GETR( chv[i].color );
    Green[i] = PPM_GETG( chv[i].color );
    Blue[i] = PPM_GETB( chv[i].color );
    }
else
    {
    Red[i] = (int) PPM_GETR( chv[i].color ) * 255 / maxval;
    Green[i] = (int) PPM_GETG( chv[i].color ) * 255 / maxval;
    Blue[i] = (int) PPM_GETB( chv[i].color ) * 255 / maxval;
    }
}
BitsPerPixel = colorstobpp( colors );

/* And make a hash table for fast lookup. */
cht = ppm_colorhisttocolorhash( chv, colors );
ppm_freecolorhist( chv );
```

```
/* All set, let's do it. */
GIFEncode(
stdout, cols, rows, interlace, 0, BitsPerPixel,
    Red, Green, Blue, GetPixel );

exit( 0 );
}
```

A few auxiliary routines follow, colorstobpp( ), which finds the number of bits, and GetPixel( ), which is passed to GIFEncode( ).

The GIFEncode( ) routines are quite general, with no particular dependence on PBM formats or libraries. One of the arguments passed to the encoder is a routine to fetch a pixel value from the image given the pixel's $x$ and $y$ coordinates. With a user-supplied routine like GetPixel( ), it is possible to support any input image encoding one wants by using a suitable routine.

```
/* compute log base 2 of colors, rounded up */
static int
colorstobpp( colors )
int colors;
    {
    int bpp;

    if ( colors <= 2 )
    bpp = 1;
    else if ( colors <= 4 )
    bpp = 2;
    else if ( colors <= 8 )
    bpp = 3;
    else if ( colors <= 16 )
    bpp = 4;
    else if ( colors <= 32 )
    bpp = 5;
    else if ( colors <= 64 )
    bpp = 6;
    else if ( colors <= 128 )
    bpp = 7;
    else if ( colors <= 256 )
    bpp = 8;
    else
    pm_error( "can't happen" );

    return bpp;
    }

/* thunk to get a pixel value from GIFEncode() */
static int
GetPixel( x, y )
```

```
int x, y;
    {
    int color;

    color = ppm_lookupcolor( cht, &pixels[y][x] );
    return color;
    }
```

## GIF Encoding

Now we examine the process of writing a GIF file. The encoder code starts with some more auxiliary routines. The first is BumpPixel( ), which handles the pixel ordering in the encoded file. It deals with interlacing by skipping around from row to row as needed, if interlacing was specified.

```
/*
 * GIFEncode( FName, GHeight, GWidth, GInterlace, Background,
 *            BitsPerPixel, Red, Green, Blue, GetPixel )
 */

#define TRUE 1
#define FALSE 0

static int Width, Height; /* overall image size */
static int curx, cury;  /* current pixel x and y position */
static long CountDown;  /* number of pixels yet to be processed */
static int Pass = 0;    /* which interlace pass */
static int Interlace;   /* whether we're interlacing */

/*
 * Bump the 'curx' and 'cury' to point to the next pixel
 */
static void
BumpPixel()
{
        /*
         * Bump the current X position
         */
        ++curx;

        /*
         * If we are at the end of a scan line, set curx back
         * to the beginning. If we are interlaced, bump the cury
         * to the appropriate spot, otherwise, just increment it.
         */
        if( curx == Width ) {
                curx = 0;

                if( !Interlace )
```

```
            ++cury;
        else {
            switch( Pass ) {

              case 0:
                 cury += 8;
                 if( cury >= Height ) {
                       ++Pass;
                       cury = 4;
                 }
                 break;

              case 1:
                 cury += 8;
                 if( cury >= Height ) {
                       ++Pass;
                       cury = 2;
                 }
                 break;

              case 2:
                 cury += 4;
                 if( cury >= Height ) {
                    ++Pass;
                    cury = 1;
                 }
                 break;

              case 3:
                 cury += 2;
                 break;
             }
        }
}
```

The incoming pixel values are returned in sequence by GIFNextPixel( ). The variable CountDown is initialized to the total number of pixels in the image (Width times Height) and counted down to track how many pixels remain. Assuming we haven't run out of pixels, GIFNextPixel fetches the current pixel and then calls BumpPixel( ) to move to the next. Following that is PutWord( ), a small routine to write a 2-byte integer in little-endian format.

```
/*
 * Return the next pixel from the image
 */
static int
GIFNextPixel( getpixel )
ifunptr getpixel;
```

```
{
        int r;

        if( CountDown == 0 )
                return EOF;

        --CountDown;

        r = ( * getpixel )( curx, cury );    /* call pixel fetcher */

        BumpPixel();            /* move to next pixel location */

        return r;
}

/*
 * Write out a word to the GIF file
 */
static void
Putword( w, fp )
int w;
FILE* fp;
{
        fputc( w & 0xff, fp );
        fputc( (w / 256) & 0xff, fp );
}
```

Now we can start to write the GIF file itself. First we set some local variables and static variables used by other routines, e.g., Width, Height, and Countdown.

```
/* write a GIF File to file fp, size GWidth by GHeight, interlace
 * GIinterlace (true/false), background color Background,
 * BitsPerPixel bits, color map in Red[], Green[], and Blue[], pixel
 * fetching routine (*GetPixel)()
 */
static void
GIFEncode( fp, GWidth, GHeight, GInterlace, Background,
           BitsPerPixel, Red, Green, Blue, GetPixel )

FILE* fp;
int GWidth, GHeight;
int GInterlace;
int Background;
int BitsPerPixel;
int Red[], Green[], Blue[];
ifunptr GetPixel;
{
        int B;
        int RWidth, RHeight;
```

```
    int LeftOfs, TopOfs;
    int Resolution;
    int ColorMapSize;
    int InitCodeSize;
    int i;

    Interlace = GInterlace;

    ColorMapSize = 1 << BitsPerPixel;

    RWidth = Width = GWidth;
    RHeight = Height = GHeight;
    LeftOfs = TopOfs = 0;
                        /* offset on screen, not often used */
    Resolution = BitsPerPixel;
                        /* resolution of original screen */
                        /* in bits */

    /*
     * Calculate number of bits we are expecting
     */
    CountDown = (long)Width * (long)Height;

    /*
     * Indicate which pass we are on (if interlace)
     */
    Pass = 0;
```

The initial code size is the number of bits per LZW encoded value, which we describe in more detail in the next section. The initial code size is the same as the number of bits per pixel, except that it must be at least 2 bits, even if the image has only 1 bit per pixel.

```
    /*
     * The initial code size
     */
    if( BitsPerPixel <= 1 )
            InitCodeSize = 2;
    else
            InitCodeSize = BitsPerPixel;

    /*
     * Set up the current x and y position
     */
    curx = cury = 0;
```

A GIF file starts with the header string "GIF87a." Our encoder doesn't use any of the new features defined in the GIF89 standard, so it recommends that we

identify the file as conforming to the older, simpler, standard. Following that are some parameters that apply to all images in the file: the screen size and height, which we make the same as the image size; the screen color resolution, which we make the same as the bits per pixel; the actual bits per pixel; and the screen background color, for which we arbitrarily choose color-map entry zero. We also indicate that a global color map is present. Other than using the color-map flag and the bits per pixel, most readers ignore all of these values.

```
/*
 * Write the Magic header
 */
fwrite( "GIF87a", 1, 6, fp );

/*
 * Write out the screen width and height
 */
Putword( RWidth, fp );
Putword( RHeight, fp );

/*
 * Indicate that there is a global color map
 */
B = 0x80;       /* Yes, there is a color map */

/*
 * OR in the resolution
 */
B |= (Resolution - 1) << 5;

/*
 * OR in the Bits per Pixel
 */
B |= (BitsPerPixel - 1);

/*
 * Write it out
 */
fputc( B, fp );

/*
 * Write out the Background color
 */
fputc( Background, fp );

/*
 * Byte of 0s (future expansion)
 */
fputc( 0, fp );
```

The GIF color map is merely a series of 3-byte RGB triples. We write the color map here.

```
/*
 * Write out the Global Color Map
 */
for( i=0; i<ColorMapSize; ++i ) {
        fputc( Red[i], fp );
        fputc( Green[i], fp );
        fputc( Blue[i], fp );
}
```

The images in a GIF file are separated by commas and end with a semi-colon, just like an English sentence, sort of. Each image starts with a comma followed by an image header. The header includes the position of the top left corner of the image relative to the edge of the screen, which we make (0,0) since the screen size is the same as the image size. It also includes a flag byte, in which we only use the interlace flag. Following that is a byte containing the minimum LZW code-word size, followed by the image itself. The file ends with a semicolon, indicating no more images.

```
/*
 * Write an Image separator
 */
fputc( ',', fp );

/*
 * Write the Image header
 */

Putword( LeftOfs, fp ); /* image offset in screen */
Putword( TopOfs, fp );
Putword( Width, fp );
Putword( Height, fp );

/*
 * Write out whether or not the image is interlaced
 */
if( Interlace )
        fputc( 0x40, fp );
else
        fputc( 0x00, fp );

/*
 * Write out the initial code size
 */
fputc( InitCodeSize, fp );
```

```
    /*
     * Go and actually compress the data
     */
    compress( InitCodeSize+1, fp, GetPixel );

    /*
     * Write out a Zero-length packet (to end the series)
     */
    fputc( 0, fp );

    /*
     * Write the GIF file terminator
     */
    fputc( ';', fp );

    /*
     * And close the file
     */
    fclose( fp );
}
```

## LZW Compression

The topic of data compression is enormous, far more than we can address here, and even the subtopic of LZW compression fills chapters. Here we concretely address the implementation of LZW as used in GIF files.

LZW compression works by building a dictionary of strings of pixels as it reads the input. Each string in the dictionary is identified by a code value, and the output of the compressor is a list of dictionary codes. The decompressor reconstructs the original pixels by looking up the strings corresponding to each code in sequence. Since a single code can correspond to a long string of pixels, the list of codes will usually be much shorter than the original input.

The compressor builds the dictionary on the fly. For example, assume that each pixel is 4 bits, with possible values of 0 through 15. So we start by setting entries 0 through 15 of our code table to single pixel strings; thus code 0 is a string consisting of a 0 pixel, code 1 is a 1 pixel, etc. (This process ensures that any possible input will have some representation as codes.) We reserve code 16 as a *Clear* code, which we will discuss shortly, and code 17 as an end of file (EOF) code, so the next unused code is number 18.

As the compressor reads pixels, it finds the longest string of pixels it can that matches an entry in the dictionary, so it always has a "tentative match" string of pixels that it has read but not yet emitted a code for. Each time the compressor reads a pixel, it does several things:

- It adds the pixel to the tentative match string and sees if the new tentative match is already in the dictionary. If so, it notes the code corresponding to the new tentative match and goes on to the next pixel.

- If the new tentative match string isn't in the dictionary, it emits the code for the old tentative match. Then it adds the new tentative match string (the old string plus the pixel just read) to the dictionary for future use, discards the old tentative match string, and starts a new tentative match string consisting only of the pixel it just read.

The compressor continues this process until it has read the entire input. At the end, it writes out the code for the final tentative match and then an EOF code.

### Building the Code Table

Since the tentative match strings are extended 1 pixel at a time, it follows that for any strings in the dictionary, all prefixes of that string are in the dictionary too. For example, if the string "3 1 4 1 6" is in the dictionary, then "3 1 4 1," "3 1 4," "3 1," and "3" will also be present and have code numbers assigned. To represent any dictionary entry, all we need to store is the value of the last pixel in the string and the code for the prefix. The dictionary might contain the entries shown in the sidebar.

| CODE | LAST | PREVIOUS |
|---|---|---|
| 100 | 6 | 87 |
| 87 | 1 | 65 |
| 65 | 4 | 58 |
| 58 | 1 | 49 |
| 3 | 3 | — |

Code 100, therefore, is the string for code 87 followed by a 6, code 87 is code 65 followed by a 1, and so forth down to code 3, which is a 3. (Recall that we initialized our dictionary with entries 0 through 15.)

In the compressor, we keep a hash table indexed by the combination of pixel value and previous code, allowing fast lookup in a fairly compact table.

### Code Sizes

The maximum code table size that GIF uses is set at 4,096 entries, which would require 12 bits to represent a code. But usually the table is nowhere near full, so a smaller number of bits suffices to represent a code. For example, if there are between 17 and 32 entries, a 5-bit code is enough; between 33 and 64 entries, a 6-bit code is enough, and so forth. Since code numbers are assigned in sequence starting at 0, the last code assigned plus 1 is the number of codes in use, which tells us the number of bits we need. We always emit a code using the smallest number of bits needed; as the code table grows, the number of bits per code grows as well.

In a reasonably large file, the code table will eventually fill up. What do we do when the table reaches 4,096 entries? We throw the entire table away and start over again with a fresh table containing just the single pixel strings, Clear, and EOF. When we throw the table away, we emit a Clear code so the decompressor that is reading the codes knows what we did. Throwing away the table may seem like a peculiar thing to do, but it turns out to be quite effective, much better than freezing the table at 4,096 entries and just using the strings already present.

The LZW compressor is based on the freely available Unix program *compress,* one of the first and by far the most most widely distributed implementations of LZW compression. Most LZW compressors that are available in source form are derived from *compress,* so if you familiarize yourself with this code, other LZW compressors will usually look familiar.

We now proceed to the actual compression code to see how the compression works in practice. First we define BITS, the maximum code size, and HSIZE, the size of the code string hash table. (The hash table has to be somewhat larger than the maximum dictionary size or the performance will become dreadful as the table fills up.)

```
#define BITS    12

#define HSIZE  5003            /* hash table size, 80% occupancy */

typedef        unsigned char   char_type;

/*
 *
 * GIF Image compression - modified 'compress'
 *
 * Based on:
 * compress.c - File compression ala IEEE Computer, June 1984.
 *
 * By Authors:
 *   Spencer W. Thomas        (decvax!harpo!utah-cs!utah-gr!thomas)
 *   Jim McKie                (decvax!mcvax!jim)
 *   Steve Davies             (decvax!vax135!petsd!peora!srd)
 *   Ken Turkowski            (decvax!decwrl!turtlevax!ken)
 *   James A. Woods           (decvax!ihnp4!ames!jaw)
 *   Joe Orost                (decvax!vax135!petsd!joe)
 *
 */

static int n_bits;              /* number of bits/code */
static int maxbits = BITS;      /* user settable max # bits/code */
static code_int maxcode;        /* maximum code, given n_bits */
static code_int maxmaxcode
    = (code_int)1 << BITS; /* should NEVER generate this code */

    /* largest code that can be represented in N bits */
#define MAXCODE(n_bits)         (((code_int) 1 << (n_bits)) - 1)
```

The hash table is the two vectors codetab[ ] and htab[ ], referenced through the macros CodeTabOf( ) and HashTabOf( ). For any particular *i,* CodeTabOf(i) is the dictionary code for entry *i,* and HashTabOf(i) is the last pixel value and the code for the prefix, packed into a long. The hash table is accessed by hashing the pixel/code pair into an integer value that is used as the index into the hash table.

```
static unsigned short codetab [HSIZE]; /* code numbers */
static count_int htab [HSIZE];    /* hash entries, pairs of codes */
#define CodeTabOf(i)    codetab[i]
#define HashTabOf(i)     htab[i]
```

```
static code_int free_ent = 0;           /* first unused entry */
static int clear_flg = 0;
                    /* about to emit first code after clear */
```

The heart of the compression code is surprisingly small. First we set up some globals used by other routines, g_init_bits, which is the number of bits initially used per output code, and g_outfile, the file to which we write the codes.

```
/*
 * compress pixels
 *
 * Algorithm:  use open addressing double hashing (no chaining)
 * on the prefix code / next pixel combination.  We do a variant
 * of Knuth's algorithm D (vol. 3, sec. 6.4) along with G. Knott's
 * relatively prime secondary probe.  Here, the modular division
 * first probe gives way to a faster exclusive-or manipulation.
 * Also do block compression with an adaptive reset, whereby the
 * code table is cleared when the compression ratio decreases but
 * after the table fills.  The variable-length output codes are
 * re-sized at this point, and a special CLEAR code is generated
 * for the decompressor.  Late addition:  construct the table
 * according to file size for noticeable speed improvement on small
 * files.  Please direct questions about this implementation to
 * ames!jaw.
 */

static int g_init_bits;         /* number of bits per output code */
static FILE* g_outfile;         /* file to write

static int ClearCode;         /* code for clear dictionary */
static int EOFCode;         /* code for end of image */

/*
 * Compress a data stream, initial code size init_bits, write
 * to file outfile, fetch input values by calling (*ReadValue)()
 */
static void
compress( init_bits, outfile, ReadValue )
int init_bits;
FILE* outfile;
ifunptr ReadValue;
{
    register long fcode;
    register code_int i /* = 0 */;
    register int c;
    register code_int ent;
    register code_int disp;
    register int hshift;
```

```
/*
 * Set up the globals:  g_init_bits - initial number of bits
 *                      g_outfile   - pointer to output file
 */
g_init_bits = init_bits;
g_outfile = outfile;
```

We set clear_flg to indicate to the output routine that we have a clear dictionary; set the clear and EOF codes to the two codes, after the ones used by the single pixel tokens; and set free_ent to be the number of the first free code number. Note that we don't actually store the single pixel codes in the table, since those codes never need to be looked up.

```
/*
 * Set up the necessary values
 */
clear_flg = 0;
maxcode = MAXCODE(n_bits = g_init_bits);
                                /* biggest code at this */
                                /* number of output bits */

ClearCode = (1 << (init_bits - 1));
EOFCode = ClearCode + 1;
free_ent = ClearCode + 2;
```

The LZW codes output by the compressor are stored into the GIF files in packets, each of which is preceded by its length. (This makes it easy to skip over an image that isn't being displayed without having to decode the whole thing.) We call char_init( ) to set up the output routines.

The first code is read by GIFNextPixel( ) into the local variable ent. At each stage in the compressor, ent contains the code for the tentative match string. Since any single pixel string is represented by a code with the same value as the pixel, this initializes ent to a single pixel string. The last initialization tasks are to set up hshift, a shift count used in the hashing scheme (it is set so that each hash value is no more than 4,096, so it doesn't run off the end of the hash table), and to initialize the dictionary by calling cl_hash( ), which sets all of the hash table entries to $-1$. We start by emitting a clear code, to make it obvious to the reader that we are starting with an empty dictionary.

```
char_init();          /* initialize byte output routines */

ent = GIFNextPixel( ReadValue );

hshift = 8;
      /* set hash code range bound for 24-bit hash vals*/
for ( fcode = (long) HSIZE;  fcode < 65536L; fcode *= 2L )
```

```
        --hshift;

    cl_hash( (count_int) HSIZE);        /* clear hash table */

    output( (code_int)ClearCode );
```

Now we enter the pixel reading and compressing loop. For each pixel we read, we create the value fcode, which is the shifted pixel value concatenated with the code number of the tentative match string. Then we see if this fcode, representing the new tentative match string, is already in the dictionary, by computing a hash value *i* and looking it up in the hash table.

```
    while ( (c = GIFNextPixel( ReadValue )) != EOF ) {

        fcode = (long) (((long) c << maxbits) + ent);
        i = (((code_int)c << hshift) ^ ent); /* xor hash code */
```

If the entry is there, we take the code for the new tentative match string, and continue to the next pixel.

```
        if ( HashTabOf (i) == fcode ) {
            ent = CodeTabOf (i);
            continue;
```

If the hash table entry is empty, the new tentative match is not in the dictionary, and we jump ahead to enter it in.

```
        } else if ( (long)HashTabOf (i) < 0 ) /* empty slot */
            goto nomatch;
```

The entry in the hash table was some other fcode, so we have a hash collision. In this case we try a sequence of other hash values until we either find the fcode, in which case we accept the tentative match string, or find a $-1$ entry, in which case we have to add the new entry to the dictionary.

```
        disp = HSIZE - i;   /* secondary hash (after G. Knott) */
        if ( i == 0 )
            disp = 1;
probe:
        if ( (i -= disp) < 0 )
            i += HSIZE;

        if ( HashTabOf (i) == fcode ) {
            ent = CodeTabOf (i);            /* found this fcode */
            continue;
        }
        if ( (long)HashTabOf (i) > 0 )
            goto probe;
```

The tentative match string isn't in the dictionary. First we output the code for the old tentative match string and make the pixel value the new tentative match. Then we try to add the new entry to the hash table—hence the dictionary. If the dictionary is full (the next code would exceed maxmaxcode), we clear the dictionary instead.

```
nomatch:
        output ( (code_int) ent );
        ent = c;
        if ( free_ent < maxmaxcode ) {
            CodeTabOf (i) = free_ent++; /* code -> hashtable */
            HashTabOf (i) = fcode;
        } else
                cl_block();
        } /* while GIFNextPixel( ... ) */
```

Having read all the pixels, we write the last code and EOF code, and we're done compressing.

```
    /*
     * Put out the final code.
     */
    output( (code_int)ent );
    output( (code_int) EOFCode );
} /* compress */
```

### Code Output

The routine output( ) actually outputs the codes for the compressed image. Since the codes are not usually a multiple of 8 bits, it has to do bit packing on the variable-length codes. The variable cur_accum keeps the bits that have been logically output but not written to the file, and cur_bits is the number of bits of valid data in cur_accum. For each output code, we OR the code into cur_accum after the bits already present. If the total number of bits now exceeds 8, we write 8-bit bytes to the file until there are less than 8 bits remaining.

```
/*****************************************************************
 * Output the given code: output(code)
 * Inputs:
 *      code:  A n_bits-bit integer.  If == -1, then EOF.
 *             This assumes that n_bits =< (long)wordsize - 1.
 * Outputs:
 *      Outputs code to the file.
 * Assumptions:
 *      Chars are 8 bits long.
 * Algorithm:
 *      Maintain a buffer in a long, output whenever there
 * are 8 bits or more buffered.
 */
```

```
static unsigned long cur_accum = 0;
                                /* partially emitted codes */
static int cur_bits = 0;        /* number of bits used */

static unsigned long masks[] = { 0x0000,
                                 0x0001, 0x0003, 0x0007, 0x000F,
                                 0x001F, 0x003F, 0x007F, 0x00FF,
                                 0x01FF, 0x03FF, 0x07FF, 0x0FFF,
                                 0x1FFF, 0x3FFF, 0x7FFF, 0xFFFF };
#if __STDC__
static void
output( code_int  code)
#else /*__STDC__*/
static void
output( code )
code_int  code;
#endif /*__STDC__*/
{
    cur_accum &= masks[ cur_bits ];

    if( cur_bits > 0 )
        cur_accum |= ((long)code << cur_bits);
    else
        cur_accum = code;

    cur_bits += n_bits;

    while( cur_bits >= 8 ) {
        char_out( (unsigned int)(cur_accum & 0xff) );
        cur_accum >>= 8;
        cur_bits -= 8;
    }
```

Each time compress( ) outputs a code, it also adds a new entry to the dictionary. For the first byte output after a clear, we set maxcode to be the maximum code value that will fit in the initial number of bits. After each entry, if the next entry won't fit in the number of bits currently being used, we increase the number of bits n_bits and adjust maxcode.

```
    /*
     * If the next entry is going to be too big for the code size,
     * then increase it, if possible.
     */
    if ( clear_flg ) {          /* first entry after a clear */

        maxcode = MAXCODE (n_bits = g_init_bits);
        clear_flg = 0;
```

```
        } else if (free_ent > maxcode ){

                ++n_bits;
                if ( n_bits == maxbits )
                    maxcode = maxmaxcode;
                else
                    maxcode = MAXCODE(n_bits);
        }
```

The code value EOFCode indicates the end of the image, so we have to flush any partially filled byte to the file.

```
    if( code == EOFCode ) {
        /*
         * At EOF, write the rest of the buffer.
         */
        while( cur_bits > 0 ) {
                char_out( (unsigned int)(cur_accum & 0xff) );
                cur_accum >>= 8;
                cur_bits -= 8;
        }

        flush_char();

        fflush( g_outfile );

        if( ferror( g_outfile ) )
                        pm_error( "error writing output file" );
    }
} /* output */
```

The routine cl_block( ) initialize the dictionary. It first calls cl_hash( ) to set all of the hash table entries to −1, meaning unused, and then resets free_ent, the first free code number to the entry after EOFCode and ClearCode, and outputs a clear code to keep the decoder in sync.

```
/*
 * Clear out the hash table
 */
static void
cl_block ()             /* table clear for block compress */
{

        cl_hash ( (count_int) HSIZE );
        free_ent = ClearCode + 2;
        clear_flg = 1;

        output( (code_int)ClearCode );
}
```

```
static void
cl_hash(hsize)          /* reset code table to -1 */
register count_int hsize;
{

        register count_int *htab_p = htab+hsize;

        register long i;
        register long m1 = -1;

        i = hsize - 16;
        do {                    /* might use Sys V memset(3) here */
                *(htab_p-16) = m1;
                *(htab_p-15) = m1;
                *(htab_p-14) = m1;
                *(htab_p-13) = m1;
                *(htab_p-12) = m1;
                *(htab_p-11) = m1;
                *(htab_p-10) = m1;
                *(htab_p-9) = m1;
                *(htab_p-8) = m1;
                *(htab_p-7) = m1;
                *(htab_p-6) = m1;
                *(htab_p-5) = m1;
                *(htab_p-4) = m1;
                *(htab_p-3) = m1;
                *(htab_p-2) = m1;
                *(htab_p-1) = m1;
                htab_p -= 16;
        } while ((i -= 16) >= 0);

        for ( i += 16; i > 0; --i )
                *--htab_p = m1;
}
```

### Writing GIF Output Packets

As we mentioned earlier, the compressed image bytes in a GIF file are stored in packets, each of which consists of a length byte followed by up to 254 data bytes. The routine char_init( ) sets up for the first packet, char_out( ) adds a byte to a packet, and flush_char( ) sends a partially or entirely full packet to the disk.

```
/*************************
 *
 * GIF Specific routines
 *
 *************************/
```

```
/*
 * Number of characters so far in this 'packet'
 */
static int a_count;

/*
 * Set up the 'byte output' routine
 */
static void
char_init()
{
        a_count = 0;
}

/*
 * Define the storage for the packet accumulator
 */
static char accum[ 256 ];

/*
 * Add a character to the end of the current packet, and if
 * it is 254 characters, flush the packet to disk.
 */
static void
char_out( c )
int c;
{
        accum[ a_count++ ] = c;
        if( a_count >= 254 )
                flush_char();
}

/*
 * Flush the packet to disk, and reset the accumulator
 */
static void
flush_char()
{
        if( a_count > 0 ) {
                fputc( a_count, g_outfile );
                fwrite( accum, 1, a_count, g_outfile );
                a_count = 0;
        }
}
```

## 17.2 READING GIF FILES

Reading a GIF file is somewhat more complex than writing one because we have to be prepared to handle various extension blocks not directly related to the image, at least enough to ignore them.

We start with miscellaneous parameters and macros.

```
/* giftoppm.c */
#include    "ppm.h"

#define MAXCOLORMAPSIZE     256

#define TRUE    1
#define FALSE   0

#define CM_RED      0       /* color map vectors */
#define CM_GREEN    1
#define CM_BLUE     2

#define MAX_LZW_BITS        12

#define INTERLACE       0x40        /* in image descriptor */
#define LOCALCOLORMAP   0x80
#define BitSet(byte, bit)   (((byte) & (bit)) == (bit))

#define ReadOK(file,buffer,len) (fread(buffer, len, 1, \
                                       file) != 0)

#define LM_to_uint(a,b)     (((b)<<8)|(a))
                             /* assemble little-endian */
```

These two structures will contain file and image header information.

```
struct {
    unsigned int    Width;
    unsigned int    Height;
    unsigned char   ColorMap[3][MAXCOLORMAPSIZE];
                                    /* global color map */
    unsigned int    BitPixel;
    unsigned int    ColorResolution;
    unsigned int    Background;
    unsigned int    AspectRatio;
} GifScreen;

struct {
    int transparent;
    int delayTime;
    int inputFlag;
    int disposal;
```

```
} Gif89 = { -1, -1, -1, 0 };

pixel   *Image = NULL;
int verbose;
int showComment;

static char usage[] = "[-verbose] [-comments] [-image N]
                       [GIFfile]";

static void ReadGIF ARGS(( FILE *fd, int imageNumber ));
static int ReadColorMap ARGS(( FILE *fd, int number,
          unsigned char buffer[3][MAXCOLORMAPSIZE] ));
static int DoExtension ARGS(( FILE *fd, int label ));
static int GetDataBlock ARGS(( FILE *fd, unsigned char  *buf ));
static int GetCode ARGS(( FILE *fd, int code_size, int flag ));
static int LZWReadByte ARGS(( FILE *fd, int flag,
          int input_code_size ));
static void ReadImage ARGS(( FILE *fd, int len, int height,
          unsigned char cmap[3][MAXCOLORMAPSIZE], int interlace,
          int ignore ));
```

We start as always by checking flags and arguments. The –image flag lets the user specify which image in a multi-image file to read.

```
void
main(argc,argv)
int argc;
char    **argv;
{
    int     argn;
    FILE        *in;
    int     imageNumber;

    ppm_init(&argc, argv);

    argn = 1;
    imageNumber = 1;
    verbose = FALSE;
    showComment = FALSE;

    while (argn < argc && argv[argn][0] ==
                '-' && argv[argn][1] != '\0') {
        if (pm_keymatch(argv[argn], "-verbose", 2)) {
            verbose = TRUE;
            showComment = TRUE;
        } else if (pm_keymatch(argv[argn], "-comments", 2)) {
            showComment = TRUE;
        } else if (pm_keymatch(argv[argn], "-image", 2)) {
            ++argn;
```

```
            if (argn == argc || sscanf(argv[argn], "%d",
                    &imageNumber) != 1)
                pm_usage(usage);
        } else {
            pm_usage(usage);
        }
        ++argn;
    }
    if (argn != argc) {
        in = pm_openr(argv[argn]);
        ++argn;
    } else {
        in = stdin;
    }
    if (argn != argc)
        pm_usage(usage);
```

Then we call ReadGIF( ) to do the actual work.

```
    ReadGIF(in, imageNumber);
    pm_close(in);
    pm_close(stdout);
    exit(0);
}
```

To process a GIF file, first we read the header, which must be either "GIF87a" or "GIF89a." Following that is the 7-byte image descriptor, which we break apart into its constituent fields. If there is a global color map, which there usually is, we read it in with ReadColorMap( ).

```
/* read imageNumber-th image from GIF file on fd, write PPM
 * to stdout
 */
static void
ReadGIF(fd, imageNumber)
FILE    *fd;
int imageNumber;
{
    unsigned char   buf[16];
    unsigned char   c;
    unsigned char   localColorMap[3][MAXCOLORMAPSIZE];
    int     useGlobalColormap;
    int     bitPixel;
    int     imageCount = 0;
    char        version[4];

    if (! ReadOK(fd,buf,6))
        pm_error("error reading magic number" );
```

```
if (strncmp(buf,"GIF",3) != 0)
    pm_error("not a GIF file" );

strncpy(version, buf + 3, 3);
version[3] = '\0';

if ((strcmp(version, "87a") != 0) &&
   (strcmp(version, "89a") != 0))
    pm_error("bad version number, not 87a or 89a" );

if (! ReadOK(fd,buf,7))
    pm_error("failed to read screen descriptor" );

GifScreen.Width           = LM_to_uint(buf[0],buf[1]);
GifScreen.Height          = LM_to_uint(buf[2],buf[3]);
GifScreen.BitPixel        = 2<<(buf[4]&0x07);
GifScreen.ColorResolution = (((buf[4]&0x70)>>3)+1);
GifScreen.Background      = buf[5];
GifScreen.AspectRatio     = buf[6];

if (BitSet(buf[4], LOCALCOLORMAP)) {
                                /* Global Colormap */
    if (ReadColorMap(fd,GifScreen.BitPixel,GifScreen.ColorMap))
        pm_error("error reading global colormap" );
}
```

The aspect ratio in the screen descriptor tells us whether the pixels in the original screen are square or rectangular. The aspect ratio is stored as an integer value $x$ such that $(x + 15)/64$ is the actual ration of $x$ size to $y$ size.

```
if (GifScreen.AspectRatio != 0 && GifScreen.AspectRatio != 49) {
    float   r;
    r = ( (float) GifScreen.AspectRatio + 15.0 ) / 64.0;
    pm_message("warning - non-square pixels;"
                    " to fix do a 'pnmscale -%cscale %g'",
        r < 1.0 ? 'x' : 'y',
        r < 1.0 ? 1.0 / r : r );
}
```

Now we read through the file looking for the image we want. Images are preceded by a comma, extension blocks by an exclamation point, and the whole file ends with a semicolon. We call DoExtension( ), below, to read an extension block.

```
for (;;) {
    if (! ReadOK(fd,&c,1))
        pm_error("EOF / read error on image data" );
```

```
if (c == ';') {     /* GIF terminator */
    if (imageCount < imageNumber)
        pm_error("only %d image%s found in file",
            imageCount, imageCount>1?"s":"" );
    return;
}

if (c == '!') {     /* Extension */
    if (! ReadOK(fd,&c,1))
        pm_error("OF / read error on extension function
                code");
    DoExtension(fd, c);
    continue;
}

if (c != ',') {     /* Not a valid start character */
    pm_message("bogus character 0x%02x, ignoring",
            (int) c );
    continue;
}
```

Now we know that we're reading an image, so we read in the 9-byte image descriptor. We read the local color map, if needed, using ReadColorMap( ) and then call ReadImage( ) to read the actual image. The arguments to ReadImage( ) include the file pointer, the *x* and *y* sizes of the image from the image header, the color map to use, the interlace flag, and a flag indicating whether we actually want to read and decode this image or just skip over it.

```
++imageCount;

if (! ReadOK(fd,buf,9))
    pm_error("couldn't read left/top/width/height");
useGlobalColormap = ! BitSet(buf[8], LOCALCOLORMAP);

bitPixel = 1<<((buf[8]&0x07)+1);

if (! useGlobalColormap) {
    if (ReadColorMap(fd, bitPixel, localColorMap))
        pm_error("error reading local colormap" );
    ReadImage(fd, LM_to_uint(buf[4],buf[5]),
            LM_to_uint(buf[6],buf[7]), localColorMap,
            BitSet(buf[8], INTERLACE), imageCount !=
            imageNumber);
} else {
    ReadImage(fd, LM_to_uint(buf[4],buf[5]),
            LM_to_uint(buf[6],buf[7]),
                    GifScreen.ColorMap,
```

```
                    BitSet(buf[8], INTERLACE), imageCount !=
                    imageNumber);
            }

        } /* read loop */
} /* ReadGIF */
```

### Reading the Color Map

We store color maps in a 3 by *N* array, where *N* is the maximum number of colors allowed in a GIF file, which is 256. The color map is just 3 byte RGB triples.

```
/* read color map from fd into buffer with number entries */
static int
ReadColorMap(fd,number,buffer)
FILE        *fd;
int     number;
unsigned char   buffer[3][MAXCOLORMAPSIZE];
{
    int     i;
    unsigned char   rgb[3];

    for (i = 0; i < number; ++i) {
        if (! ReadOK(fd, rgb, sizeof(rgb)))
            pm_error("bad colormap" );

        buffer[CM_RED][i] = rgb[0] ;
        buffer[CM_GREEN][i] = rgb[1] ;
        buffer[CM_BLUE][i] = rgb[2] ;
    }
    return FALSE;
}
```

### GIF Extensions

GIF files can contain a variety of extension types, most of which have little to do with the actual image. Types include:

**Plain Text** Text strings to be overlaid on the image.

**Application** Unspecified data for applications that exchange GIF files.

**Comment** A string describing the image.

**Graphic Control** Hints about how to display the image, e.g., how long to display it, whether to keep it around after displaying it, or whether some pixels should be transparent.

We decode most of these extensions, but do no more than mention that they have appeared.

```
static int
DoExtension(fd, label)
FILE    *fd;
int label;
{
    static char buf[256];
    char        *str;

    switch (label) {
    case 0x01:      /* Plain Text Extension */
        str = "Plain Text Extension";
        break;
    case 0xff:      /* Application Extension */
        str = "Application Extension";
        break;
    case 0xfe:      /* Comment Extension */
        str = "Comment Extension";
        while (GetDataBlock(fd, (unsigned char*) buf) != 0) {
            if (showComment)
                pm_message("gif comment: %s", buf );
        }
        return FALSE;
    case 0xf9:      /* Graphic Control Extension */
        str = "Graphic Control Extension";
        (void) GetDataBlock(fd, (unsigned char*) buf);
        Gif89.disposal    = (buf[0] >> 2) & 0x7;
        Gif89.inputFlag   = (buf[0] >> 1) & 0x1;
        Gif89.delayTime   = LM_to_uint(buf[1],buf[2]);
        if ((buf[0] & 0x1) != 0)
            Gif89.transparent = buf[3];

        while (GetDataBlock(fd, (unsigned char*) buf) != 0)
            ;
        return FALSE;
    default:
        str = buf;
        sprintf(buf, "UNKNOWN (0x%02x)", label);
        break;
    }

    pm_message("got a '%s' extension", str );

    while (GetDataBlock(fd, (unsigned char*) buf) != 0)
        ;

    return FALSE;
}
```

## Decoding the Compressed Image

The compressed image pixels are stored in packets consisting of a count byte followed by data. The routine GetDataBlock( ) reads a packet.

```
int ZeroDataBlock = FALSE;

/* get a packet from file fd into buf[] */
static int
GetDataBlock(fd, buf)
FILE        *fd;
unsigned char   *buf;
{
    unsigned char   count;

    if (! ReadOK(fd,&count,1)) {
        pm_message("error in getting DataBlock size" );
        return -1;
    }

    ZeroDataBlock = count == 0;

    if ((count != 0) && (! ReadOK(fd, buf, count))) {
        pm_message("error in reading DataBlock" );
        return -1;
    }

    return count;
}
```

Individual code words are read by GetCode( ), which treats a block of data as an array of bits, with curbit pointing to the next bit to retrieve, and lastbit being the number of bits in the buffer. When most of the bits in the buffer have been consumed, GetCode slides the last 2 bytes in the buffer up to the front and reads another packet after those 2 bytes. (Retaining the 2 bytes handles cases in which a code is split between packets.) Then it updates curbit, which must have been pointing to a bit within the 2 bytes that were moved, to the new bit's position, and lastbit and lastbyte to reflect the new amount of valid data.

Once we're sure we have enough bits remaining in the buffer, we pick out as many bits as are required, assemble them into an int, and return them.

```
/* read a code from file on fd, size in bits code_size.  Non-zero
 * flag means to initialize, not read
 */
static int
GetCode(fd, code_size, flag)
FILE    *fd;
int code_size;
int flag;
```

```
{
    static unsigned char    buf[280];
    static int  curbit,          /* current bit number in buf[] */
                lastbit,
                 /* number of bits in buf[] */
                done,
                 /* hit file EOF */
                last_byte;
                 /* last valid byte # in buffer */
    int          i, j, ret;
    unsigned char       count;

    if (flag) {                    /* prepare to start reading
        curbit = 0;
        lastbit = 0;
        done = FALSE;
        return 0;
    }

    if ( (curbit+code_size) >= lastbit) { /* need more bits ? */
        if (done) {
            if (curbit >= lastbit)
                pm_error("ran off the end of my bits" );
            return -1;
        }
        buf[0] = buf[last_byte-2];
                          /* slide last two bytes to the front */
        buf[1] = buf[last_byte-1];

                         /* read new data after those two bytes */
        if ((count = GetDataBlock(fd, &buf[2])) == 0)
            done = TRUE;
        last_byte = 2 + count;
                                  /* update count of valid bytes */
        curbit = (curbit - lastbit) + 16;
                                  /* account for slide */
        lastbit = (2+count)*8 ;
    }

    ret = 0;                      /* pick out bits one at a time */
    for (i = curbit, j = 0; j < code_size; ++i, ++j)
        ret |= ((buf[ i / 8 ] & (1 << (i % 8))) != 0) << j;

    curbit += code_size;          /* update bit pointer */

    return ret;
}
```

### Decoding LZW Data

Armed with routines to read in the variable-length code words, we can now proceed to the actual LZW decoding. Decoding depends on the following observation: since the decoder reconstructs the same stream of pixels that the encoder started with, the decoder can construct the same dictionary. If, as we read and decode the LZW codes, we add entries to the dictionary as we go, then the decoder's dictionary will always be in sync with the encoder's and the codes encountered in the input stream will always be in the decoder's dictionary.

The way that entries are added to the dictionary is quite simple. Each new entry consists of an existing entry, followed by the next pixel from the original image. So if the compressed data contains the code for "2 3 4" followed by the code for "7 8 9," the 7 must have followed the "2 3 4," so the next string to put in the dictionary is "2 3 4 7." Since the dictionary strings are built up one character at a time, the obvious way to keep the dictionary is essentially as a linked list of characters. We keep two arrays, vals[ ] and next[ ]. For any code *i,* vals[i] is the last pixel in the string for *i,* and next[i] is the code for the string with the preceding pixels. If next[i] is less than the Clear code, it is one of the original pixel entries in the dictionary, hence the last link in the list and the first pixel in the string. Since the pixels are linked together in reverse order (end of the string to the beginning) we also keep a stack that buffers a string to be returned.

At each stage, if there are any pixels on the stack, we pop the top pixel from the stack. Otherwise, we read another code, push the pixels corresponding to that code on the stack, and add a new dictionary code corresponding to the *previous* code followed by the first pixel in the expansion of the current code. In LZWReadByte( ), the variable oldcode holds the previous code on which the next dictionary entry will be based, and firstcode holds the pixel that will be appended to it to make the new string.

```
/* read a pixel from file fd.  If flag is set, initialize routines
 * starting with codes of input_code_size bits
 */
static int
LZWReadByte(fd, flag, input_code_size)
FILE    *fd;
int flag;
int input_code_size;
{
    static int  fresh = FALSE;
    int     code, incode;
    static int  code_size, set_code_size;
    static int  max_code, max_code_size;
    static int  firstcode, oldcode;
    static int  clear_code, end_code;
#ifdef MSDOS
        /* put these large vectors in other segments on DOS */
    static unsigned short __far next[1<< MAX_LZW_BITS];
    static unsigned char __far vals[1<< MAX_LZW_BITS];
```

```
    static unsigned char __far stack[1<<(MAX_LZW_BITS+1)], *sp;
#else
    static unsigned short next[1<< MAX_LZW_BITS];
    static unsigned char vals[1<< MAX_LZW_BITS];
    static unsigned char stack[1<<(MAX_LZW_BITS+1)], *sp;
#endif
    register int    i;
```

The first call to LZWReadByte( ) has the flag set to indicate that it's for initialization. We set set_code_size (the size of the pixels), code_size (the initial number of bits per code word), clear_code (the code that says to clear the dictionary), end_code (the code that marks logical EOF), max_code (the last code assigned), and max_code_size (the next code value that will require increasing the code size). We also initialize GetCode( ), install the single pixel strings in the dictionary, clear out the rest of the dictionary, and initialize the stack pointer sp.

```
if (flag) {                     /* initialization call */
    set_code_size = input_code_size;
    code_size = set_code_size+1;
    clear_code = 1 << set_code_size ;
    end_code = clear_code + 1;
    max_code = clear_code+2;
    max_code_size = 2*clear_code;

    GetCode(fd, 0, TRUE);

    fresh = TRUE;           /* note next is first decode call */

            /* single pixel strings */
    for (i = 0; i < clear_code; ++i) {
        next[i] = 0;
        vals[i] = i;
    }
            /* clear the rest of the dictionary, just in case */
    for (; i < (1<<MAX_LZW_BITS); ++i)
        next[i] = vals[0] = 0;

    sp = stack;

    return 0;
```

On the first call to retrieve a pixel, we read code words until we get one that is not a clear code. Since the dictionary contains only single pixel strings, this code must be the first pixel. We also remember it as oldcode.

```
} else if (fresh) {             /* first decode call */
    fresh = FALSE;
    do {
```

```
            firstcode = oldcode =
                GetCode(fd, code_size, FALSE);
        } while (firstcode == clear_code);
        return firstcode;
    }
```

Here we start the normal operation of the decoder. If there are still pixels on the stack, we pop the next pixel and return.

```
    if (sp > stack) /* still working through a previously */
        return *--sp; /* decoded string */
```

Now the stack is empty, so we have to do some serious decoding. First we handle two special cases. If we receive a clear code, we reinitialize the dictionary the same way that we did above. Then we read another code, which must represent a single pixel string, and return it.

```
    while ((code = GetCode(fd, code_size, FALSE)) >= 0) {

        if (code == clear_code) { /* clear out the dictionary */
            for (i = 0; i < clear_code; ++i) {
                next[i] = 0;
                vals[i] = i;
            }
            for (; i < (1<<MAX_LZW_BITS); ++i)
                next[i] = vals[i] = 0;
            code_size = set_code_size+1;
            max_code_size = 2*clear_code;
            max_code = clear_code+2;
            sp = stack;
            firstcode = oldcode =
                    GetCode(fd, code_size, FALSE);
            return firstcode;
```

At the end of the image we get an EOF code. In that case, we discard any extra code data if there is any (the variable ZeroDataBlock notes that the end of the packetized data was already read), and return −2, indicating end of image.

```
        } else if (code == end_code) {          /* end of image */
            int     count;
            unsigned char   buf[260];

            if (ZeroDataBlock)
                return -2;

            while ((count = GetDataBlock(fd, buf)) > 0)
                ;                       /* discard input data */
```

```
        if (count != 0)
            pm_message("missing EOD in data stream
                        (common occurrence)");
        return -2;
    }
```

We decode the code we just read. If the code is equal to max_code, the encoder has gotten a little ahead of us; the code is the one that we are about to add, so we push the old firstcode and use the previous code.

```
    incode = code;

    if (code >= max_code) {
                            /* defensive special case for preview */
                            /* of about to be inserted code */
        *sp++ = firstcode;
        code = oldcode;
    }
```

We run down the link-list representing this string, pushing all of the pixels onto the stack. When a code is less than clear_code, it is one of the original single-pixel strings, so we know we've hit the front of the string.

```
    while (code >= clear_code) { /* run through dict pushing */
        *sp++ = vals[code];     /* entire string on the stack */
        if (code == next[code])
            pm_error("circular table entry BIG ERROR");
        code = next[code];
    }
```

We push the last pixel in the list (i.e., the first pixel in the string) and remember it as firstcode.

The next task is to add the new dictionary entry. We have oldcode and firstcode, so we enter them in the dictionary arrays, assuming the dictionary is not full. If this code reaches max_code_size, it's time to increase the code size another bit, so we do.

```
    *sp++ = firstcode = vals[code];
                                    /* first pixel in dict string */

    if ((code = max_code) <(1<<MAX_LZW_BITS)) {
                                                    /* add to */
                    next[code] = oldcode;       /* dictionary */

        vals[code] = firstcode;
        ++max_code;
        if ((max_code >= max_code_size) &&
                                                /* increase code size */
```

```
                (max_code_size < (1<<MAX_LZW_BITS))) {
                max_code_size *= 2;
                ++code_size;
            }
        }
```

Now we've pushed the decoded string and entered the new dictionary entry. Now we return the top pixel from the stack.

```
        oldcode = incode;

        if (sp > stack)
            return *--sp;
    } /* while GetCode() */
    return code;        /* end of codes */
} /* LZWReadByte */
```

Using the LZW decoder, it is now straightforward to decode an image. The first byte in each image is the initial code size, so we first read that byte. Then, if the ignore flag is set, meaning this isn't the image we want, we read and discard all the pixels.

```
static void
ReadImage(fd, len, height, cmap, interlace, ignore)
FILE    *fd;
int len, height;
unsigned char   cmap[3][MAXCOLORMAPSIZE];
int interlace, ignore;
{
    unsigned char   c;
    int     v;
    int     xpos = 0, ypos = 0, pass = 0;
    pixel       **image;

    /*
    **  Initialize the Compression routines
    */
    if (! ReadOK(fd,&c,1))        /* get initial code size */
        pm_error("EOF / read error on image data" );

    if (LZWReadByte(fd, TRUE, c) < 0)   /* initialize LZW */
        pm_error("error reading image" );

    /*
    **  If this is an "uninteresting picture" ignore it.
    */
    if (ignore) {
        if (verbose)
            pm_message("skipping image..." );
```

```
        while (LZWReadByte(fd, FALSE, c) >= 0)
            ;
        return;
    }
```

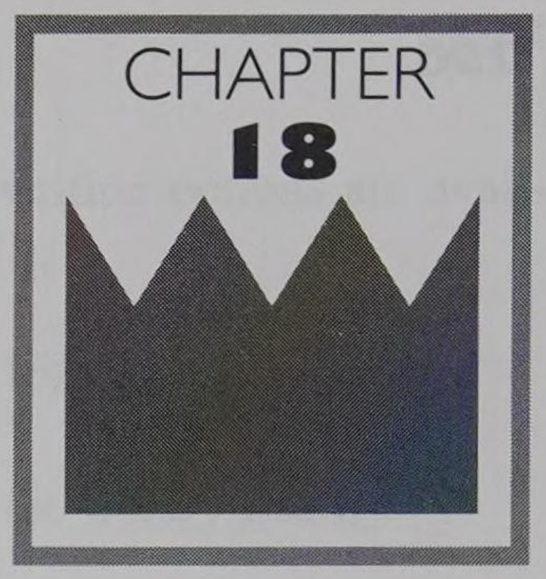
Here we know this is the image we want, so we allocate the PPM storage for it and start reading and storing the pixels row-by-row. If the interlace flag is set, we have to scramble the rows around appropriately.

```
    if ((image = ppm_allocarray(len, height)) == NULL)
        pm_error("couldn't alloc space for image" );

    if (verbose)
        pm_message("reading %d by %d%s GIF image",
            len, height, interlace ? " interlaced" : "" );

    while ((v = LZWReadByte(fd,FALSE,c)) >= 0 ) {
            /* look up pixel in color map and store it away */
        PPM_ASSIGN(image[ypos][xpos], cmap[CM_RED][v],
                   cmap[CM_GREEN][v], cmap[CM_BLUE][v]);

        ++xpos;
        if (xpos == len) {
            xpos = 0;
            if (interlace) {  /* handle interlaced row order */
                switch (pass) {
                case 0:
                case 1:
                    ypos += 8; break;
                case 2:
                    ypos += 4; break;
                case 3:
                    ypos += 2; break;
                }
                if (ypos >= height) {
                    ++pass;
                    switch (pass) {
                    case 1:
                        ypos = 4; break;
                    case 2:
                        ypos = 2; break;
                    case 3:
                        ypos = 1; break;
                    default:
                        goto fini;
                    }
                }
```

```
            } else {
                ++ypos;          /* no interlace, read in order */
            }
        }
        if (ypos >= height)
            break;
    }
```

At long last we've read the entire image. We check for and ignore any garbage pixels at the end of the image, write out the PPM pixels, and we are done.

```
fini:
    if (LZWReadByte(fd,FALSE,c)>=0)
        pm_message("too much input data, ignoring extra...");
    if (verbose)
        pm_message("writing output");
    ppm_writeppm(stdout, image, len, height, (pixval) 255, 0 );
}
```




# TIFF

Tagged Image File Format (TIFF) is one of the most popular formats for bit-mapped files. Unfortunately, it is also by far the most difficult format to handle because it has dozens and dozens of different subformats. Vendors can even register their own private and undocumented formats and produce TIFF files that only they can read.

## 18.1 TIFF SUBSETS AND CLASSES

The TIFF specification has gone through many revisions; the most recent version is 6.0. Earlier versions of the spec defined TIFF "classes," which used only a subset of TIFF features. For example, Class B was for bi-level images with single-bit pixels, Class G for grayscale, Class P for color images with color maps, and Class R for full-color images. The 6.0 specification simplifies the classification to Baseline TIFF, which all readers have to support, and TIFF Extensions, which are optional. Baseline TIFF still has a lot of options that support all the old classes with images stored either uncompressed, with PackBits, or, for bi-level images, CCITT run-length encoding. The TIFF library code we display here deals only with the most common Baseline features. (The full version on the companion diskette handles nearly all known TIFF formats.)

TIFF files start with a short header that identifies the byte order used in the file (TIFF files can be either big-endian or little-endian). The header contains the magic number 42, to identify the file as a TIFF file, and a pointer to the file position of the Image File Directory, or IFD. The IFD contains a set of 12-byte entries. Each entry has a tag to identify the type of the entry, a Type (character, short, floating point, etc.), the number of values in the entry, and a pointer to the file offset of the entry. The TIFF standard defines over 100 tag types. For example, consider the entry for the color map in a file using 8-bit pixels. The tag has type 320, meaning it's the color map. The type field is 8, meaning 16-bit

short values (TIFF uses 16-bit color map components). The count field is 768, since the color map has 256 entries of three components apiece, a total of 768 values. The offset is the place where the color map starts, which can be anywhere in the file. As a special case, if the data for an entry fits into 4 bytes or less, it is actually stored in the offset field of the IFD entry itself. The IFD may contain a pointer to another IFD, which describes another image stored in the same file. Baseline TIFF doesn't support that feature, though.

The IFD contains fields that describe the image, including photometric interpretation (whether a 0 pixel is black or white), the compression scheme, number of rows, number of columns, desired physical size of the image in inches or centimeters, and the location of the file where the actual pixel data is stored. Images are usually stored in several strips, and each strip contains several rows. The strips are typically sized to be about 8K apiece to make them easier to handle on systems with limited memory. Some TIFF files further subdivide the image vertically into tiles, although baseline TIFF doesn't support that.

## 18.2 THE TIFF LIBRARY

We first look at a general-purpose TIFF library that reads and writes TIFF files. Then we build some small applications that call the libraries and turn TIFF files into PBM, PGM, or PPM files and back.

### Using the TIFF Library

The TIFF library is, generally speaking, modeled after the familiar standard I/O library. To read a TIFF file, the steps are:

- Open the file with TIFFOpen( ).
- Retrieve fields of interest with TiffGetField( ). Each call returns one field from the IFD.
- Read the pixels from the image. There are several different pixel reading routines that one might call depending on how much processing the library needs to do versus how much the application does. The easiest to use is TIFFReadScanline( ), which reads one row from the image, and which the application usually calls in a loop to retrieve all of the rows.
- Close the file with TIFFClose( ).

Writing a file works in much the same way:

- Open the file with TIFFOpen( ).
- Set the fields of interest with TiffSetField( ). Each call sets one field in the IFD.

- Write the pixels into the image. The same pixel-writing options are available, with the simplest being TIFFWriteScanline( ).
- Close the file with TIFFClose( ) to write out the IFD.

With this brief overview, we turn to the structure of the library itself.

## Library Headers

Our first include file is tiff.h, which defines values and data structures found in all TIFF files. It starts by defining the TIFF file header, which contains the magic number that determines the file's byte order (the 2 bytes II for little-endian or MM for big-endian), the equally magic version number 42, and the pointer to the IFD.

```
/* header file tiff.h  */
#ifndef _TIFF_
#define _TIFF_
/*
 * Tag Image File Format (TIFF)
 *
 * Based on Rev 6.0 from:
 *    Developer's Desk
 *    Aldus Corporation
 *    411 First Ave. South
 *    Suite 200
 *    Seattle, WA  98104
 *    206-622-5500
 */
#define TIFF_VERSION    42

#define TIFF_BIGENDIAN      0x4d4d
#define TIFF_LITTLEENDIAN   0x4949

typedef struct {
    unsigned short tiff_magic;
                            /* magic number (defines byte order) */
    unsigned short tiff_version;         /* TIFF version number */
    unsigned long  tiff_diroff;
                                /* byte offset to first directory */
} TIFFHeader;
```

### Image File Directories

We define the structure of an Image File directory and define constants for all of the tags and conventional values used in IFDs.

```
/*
 * TIFF Image File Directories are comprised of
 * a table of field descriptors of the form shown
 * below.  The table is sorted in ascending order
 * by tag.  The values associated with each entry
 * are disjoint and may appear anywhere in the file
 * (so long as they are placed on a word boundary).
 *
 * If the value is 4 bytes or less, then it is placed
 * in the offset field to save space.  If the value
 * is less than 4 bytes, it is left-justified in the
 * offset field.
 */
typedef struct {
    unsigned short tdir_tag;    /* see below */
    unsigned short tdir_type;   /* datatype; see below */
    unsigned long  tdir_count;
                        /* number of items; length in spec */
    unsigned long  tdir_offset; /* byte offset to field data */
} TIFFDirEntry;

/*
 * NB: In the comments below,
 *  - items marked with a + are made obsolete by revision 5.0,
 *  - items marked with a ! are introduced in revision 6.0.
 *  - items marked with a $ are obsoleted by revision 6.0.
 */
```

These are the data type codes found in tdir_type. Baseline TIFF only handles BYTE, ASCII, SHORT, LONG, and RATIONAL types, though a reader should at least recognize and reject other types.

```
/*
 * Tag datatype information.
 *
 * Note: RATIONALs are the ratio of two 32-bit integer values.
 */
typedef enum {
    TIFF_NOTYPE = 0,     /* placeholder */
    TIFF_BYTE   = 1,     /* 8-bit unsigned integer */
    TIFF_ASCII  = 2,     /* 8-bit bytes w/ last byte null */
    TIFF_SHORT  = 3,     /* 16-bit unsigned integer */
    TIFF_LONG   = 4,     /* 32-bit unsigned integer */
    TIFF_RATIONAL   = 5,     /* 64-bit unsigned fraction */
    TIFF_SBYTE  = 6,     /* !8-bit signed integer */
    TIFF_UNDEFINED  = 7,     /* !8-bit untyped data */
    TIFF_SSHORT = 8,     /* !16-bit signed integer */
    TIFF_SLONG  = 9,     /* !32-bit signed integer */
    TIFF_SRATIONAL  = 10,    /* !64-bit signed fraction */
    TIFF_FLOAT  = 11,    /* !32-bit IEEE floating point */
```

```
    TIFF_DOUBLE = 12    /* !64-bit IEEE floating point */
} TIFFDataType;
```

Here we list all of the tags. In many cases, there are only a few valid values for a field, in which case the valid values are given names as well. For example, the valid values in a TIFFTAG_SUBFILETYPE field are FILETYPE_REDUCEDIMAGE, FILETYPE_PAGE, and FILETYPE_MASK.

```
/*
 * TIFF Tag Definitions.
 */
#define TIFFTAG_SUBFILETYPE     254 /* subfile data descriptor */
#define     FILETYPE_REDUCEDIMAGE   0x1
                                    /* reduced resolution version */
#define     FILETYPE_PAGE       0x2 /* one page of many */
#define     FILETYPE_MASK       0x4 /* transparency mask */
#define TIFFTAG_OSUBFILETYPE        255
                                    /* +kind of data in subfile */
#define     OFILETYPE_IMAGE     1   /* full resolution image data */
#define     OFILETYPE_REDUCEDIMAGE  2  /* reduced size image data */
#define     OFILETYPE_PAGE      3   /* one page of many */
#define TIFFTAG_IMAGEWIDTH      256 /* image width in pixels */
#define TIFFTAG_IMAGELENGTH     257 /* image height in pixels */
#define TIFFTAG_BITSPERSAMPLE       258
                                    /* bits per channel (sample) */
#define TIFFTAG_COMPRESSION     259 /* data compression technique */
#define     COMPRESSION_NONE        1   /* dump mode */
#define     COMPRESSION_CCITTRLE    2
                                /* CCITT modified Huffman RLE */
#define     COMPRESSION_CCITTFAX3   3
                                /* CCITT Group 3 fax encoding */
#define     COMPRESSION_CCITTFAX4   4
                                /* CCITT Group 4 fax encoding */
#define     COMPRESSION_LZW     5   /* Lempel-Ziv  & Welch */
#define     COMPRESSION_JPEG        6   /* !JPEG compression */
#define     COMPRESSION_NEXT        32766 /* NeXT 2-bit RLE */
#define     COMPRESSION_CCITTRLEW   32771 /* #1 w/ word alignment */
#define     COMPRESSION_PACKBITS    32773 /* Macintosh RLE */
#define     COMPRESSION_THUNDERSCAN 32809 /* ThunderScan RLE */
#define TIFFTAG_PHOTOMETRIC     262 /* photometric interpretation */
#define     PHOTOMETRIC_MINISWHITE  0   /* min value is white */
#define     PHOTOMETRIC_MINISBLACK  1   /* min value is black */
#define     PHOTOMETRIC_RGB     2   /* RGB color model */
#define     PHOTOMETRIC_PALETTE     3   /* color map indexed */
#define     PHOTOMETRIC_MASK        4   /* $holdout mask */
#define     PHOTOMETRIC_SEPARATED   5   /* !color separations */
#define     PHOTOMETRIC_YCBCR       6   /* !CCIR 601 */
#define     PHOTOMETRIC_CIELAB      8   /* !1976 CIE L*a*b* */
#define TIFFTAG_THRESHHOLDING       263
                                    /* +thresholding used on data */
```

```
#define     THRESHHOLD_BILEVEL      1  /* b&w art scan */
#define     THRESHHOLD_HALFTONE     2  /* or dithered scan */
#define     THRESHHOLD_ERRORDIFFUSE 3  /* usually floyd-steinberg */
#define TIFFTAG_CELLWIDTH       264 /* +dithering matrix width */
#define TIFFTAG_CELLLENGTH      265 /* +dithering matrix height */
#define TIFFTAG_FILLORDER       266 /* data order within a byte */
#define     FILLORDER_MSB2LSB       1
                                  /* most significant -> least */
#define     FILLORDER_LSB2MSB       2
                                  /* least significant -> most */
#define TIFFTAG_DOCUMENTNAME        269
                                  /* name of doc. image is from */
#define TIFFTAG_IMAGEDESCRIPTION    270 /* info about image */
#define TIFFTAG_MAKE            271 /* scanner manufacturer name */
#define TIFFTAG_MODEL           272 /* scanner model name/number */
#define TIFFTAG_STRIPOFFSETS        273
                                  /* offsets to data strips */
#define TIFFTAG_ORIENTATION     274 /* +image orientation */
#define     ORIENTATION_TOPLEFT     1   /* row 0 top, col 0 lhs */
#define     ORIENTATION_TOPRIGHT    2   /* row 0 top, col 0 rhs */
#define     ORIENTATION_BOTRIGHT    3
                                   /* row 0 bottom, col 0 rhs */
#define     ORIENTATION_BOTLEFT     4
                                   /* row 0 bottom, col 0 lhs */
#define     ORIENTATION_LEFTTOP     5  /* row 0 lhs, col 0 top */
#define     ORIENTATION_RIGHTTOP    6  /* row 0 rhs, col 0 top */
#define     ORIENTATION_RIGHTBOT    7
                                   /* row 0 rhs, col 0 bottom */
#define     ORIENTATION_LEFTBOT     8  /* row 0 lhs, col 0 bottom */
#define TIFFTAG_SAMPLESPERPIXEL     277 /* samples per pixel */
#define TIFFTAG_ROWSPERSTRIP        278 /* rows per strip of data */
#define TIFFTAG_STRIPBYTECOUNTS     279
                                     /* bytes counts for strips */
#define TIFFTAG_MINSAMPLEVALUE      280  /* +minimum sample value */
#define TIFFTAG_MAXSAMPLEVALUE      281  /* +maximum sample value */
#define TIFFTAG_XRESOLUTION     282 /* pixels/resolution in x */
#define TIFFTAG_YRESOLUTION     283 /* pixels/resolution in y */
#define TIFFTAG_PLANARCONFIG        284 /* storage organization */
#define     PLANARCONFIG_CONTIG     1  /* single image plane */
#define     PLANARCONFIG_SEPARATE   2  /* separate planes of data */
#define TIFFTAG_PAGENAME        285 /* page name image is from */
#define TIFFTAG_XPOSITION       286 /* x page offset of image lhs */
#define TIFFTAG_YPOSITION       287 /* y page offset of image lhs */
#define TIFFTAG_FREEOFFSETS     288 /* +byte offset to free block */
#define TIFFTAG_FREEBYTECOUNTS      289  /* +sizes of free blocks */
#define TIFFTAG_GRAYRESPONSEUNIT    290
                                  /* $grayscale curve accuracy */
#define     GRAYRESPONSEUNIT_10S    1  /* tenths of a unit */
```

```
#define     GRAYRESPONSEUNIT_100S   2  /* hundredths of a unit */
#define     GRAYRESPONSEUNIT_1000S  3  /* thousandths of a unit */
#define     GRAYRESPONSEUNIT_10000S 4
                                    /* ten-thousandths of a unit */
#define     GRAYRESPONSEUNIT_100000S    5 /* hundred-thousandths */
#define TIFFTAG_GRAYRESPONSECURVE   291
                                    /* $grayscale response curve */
#define TIFFTAG_GROUP3OPTIONS       292 /* 32 flag bits */
#define     GROUP3OPT_2DENCODING    0x1 /* two-dimensional coding */
#define     GROUP3OPT_UNCOMPRESSED  0x2 /* data not compressed */
#define     GROUP3OPT_FILLBITS      0x4 /* fill to byte boundary */
#define TIFFTAG_GROUP4OPTIONS       293 /* 32 flag bits */
#define     GROUP4OPT_UNCOMPRESSED  0x2 /* data not compressed */
#define TIFFTAG_RESOLUTIONUNIT      296 /* units of resolutions */
#define     RESUNIT_NONE        1   /* no meaningful units */
#define     RESUNIT_INCH        2   /* english */
#define     RESUNIT_CENTIMETER      3   /* metric */
#define TIFFTAG_PAGENUMBER      297 /* page numbers of multi-page */
#define TIFFTAG_COLORRESPONSEUNIT   300 /* $color curve accuracy */
#define     COLORRESPONSEUNIT_10S   1   /* tenths of a unit */
#define     COLORRESPONSEUNIT_100S  2   /* hundredths of a unit */
#define     COLORRESPONSEUNIT_1000S 3   /* thousandths of a unit */
#define     COLORRESPONSEUNIT_10000S    4
                                    /* ten-thousandths of a unit */
#define     COLORRESPONSEUNIT_100000S   5 /* hundred-thousandths */
#define TIFFTAG_TRANSFERFUNCTION    301 /* !colorimetry info */
#define TIFFTAG_SOFTWARE        305 /* name & release */
#define TIFFTAG_DATETIME        306 /* creation date and time */
#define TIFFTAG_ARTIST          315 /* creator of image */
#define TIFFTAG_HOSTCOMPUTER        316 /* machine where created */
#define TIFFTAG_PREDICTOR       317 /* prediction scheme w/ LZW */
#define TIFFTAG_WHITEPOINT      318 /* image white point */
#define TIFFTAG_PRIMARYCHROMATICITIES   319
                                    /* !primary chromaticities */
#define TIFFTAG_COLORMAP        320 /* RGB map for palette image */
#define TIFFTAG_HALFTONEHINTS       321 /* !highlight+shadow info */
#define TIFFTAG_TILEWIDTH       322 /* !rows/data tile */
#define TIFFTAG_TILELENGTH      323 /* !cols/data tile */
#define TIFFTAG_TILEOFFSETS     324 /* !offsets to data tiles */
#define TIFFTAG_TILEBYTECOUNTS      325 /* !byte counts for tiles */
#define TIFFTAG_BADFAXLINES     326 /* lines w/ wrong pixel count */
#define TIFFTAG_CLEANFAXDATA        327 /* regenerated line info */
#define     CLEANFAXDATA_CLEAN      0   /* no errors detected */
#define     CLEANFAXDATA_REGENERATED    1
                                /* receiver regenerated lines */
#define     CLEANFAXDATA_UNCLEAN    2
                                    /* uncorrected errors exist */
#define TIFFTAG_CONSECUTIVEBADFAXLINES  328
                                /* max consecutive bad lines */
```

```
#define TIFFTAG_INKSET          332 /* !inks in separated image */
#define     INKSET_CMYK         1   /* !cyan-magenta-yellow-black */
#define TIFFTAG_INKNAMES        333 /* !ascii names of inks */
#define TIFFTAG_DOTRANGE        336 /* !0% and 100% dot codes */
#define TIFFTAG_TARGETPRINTER       337 /* !separation target */
#define TIFFTAG_EXTRASAMPLES        338
                                   /* !info about extra samples */
#define     EXTRASAMPLE_UNSPECIFIED 0   /* !unspecified data */
#define     EXTRASAMPLE_ASSOCALPHA  1
                                    /* !associated alpha data */
#define     EXTRASAMPLE_UNASSALPHA  2
                                    /* !unassociated alpha data */
#define TIFFTAG_SAMPLEFORMAT        339 /* !data sample format */
#define     SAMPLEFORMAT_INT        1  /* !signed integer data */
#define     SAMPLEFORMAT_UINT       2  /* !unsigned integer data */
#define     SAMPLEFORMAT_IEEEFP     3
                                    /* !IEEE floating point data */
#define     SAMPLEFORMAT_VOID       4  /* !untyped data */
#define TIFFTAG_SMINSAMPLEVALUE     340
                                    /* !variable MinSampleValue */
#define TIFFTAG_SMAXSAMPLEVALUE     341
                                    /* !variable MaxSampleValue */
#define TIFFTAG_JPEGPROC        512 /* !JPEG processing algorithm */
#define     JPEGPROC_BASELINE       1   /* !baseline sequential */
#define     JPEGPROC_LOSSLESS       14
                                    /* !Huffman coded lossless */
#define TIFFTAG_JPEGIFOFFSET        513 /* !pointer to SOI marker */
#define TIFFTAG_JPEGIFBYTECOUNT     514 /* !JFIF stream length */
#define TIFFTAG_JPEGRESTARTINTERVAL 515
                                    /* !restart interval length */
#define TIFFTAG_JPEGLOSSLESSPREDICTORS  517
                                    /* !lossless proc predictor */
#define TIFFTAG_JPEGPOINTTRANSFORM  518
                                   /* !lossless point transform */
#define TIFFTAG_JPEGQTABLES     519 /* !Q matrices offsets */
#define TIFFTAG_JPEGDCTABLES        520 /* !DCT table offsets */
#define TIFFTAG_JPEGACTABLES        521
                                     /* !AC coefficient offsets */
#define TIFFTAG_YCBCRCOEFFICIENTS   529
                                     /* !RGB -> YCbCr transform */
#define TIFFTAG_YCBCRSUBSAMPLING    530
                                   /* !YCbCr subsampling factors */
#define TIFFTAG_YCBCRPOSITIONING    531
                                   /* !subsample positioning */
#define     YCBCRPOSITION_CENTERED  1
                                   /* !as in PostScript Level 2 */
#define     YCBCRPOSITION_COSITED   2   /* !as in CCIR 601-1 */
#define TIFFTAG_REFERENCEBLACKWHITE 532 /* !colorimetry info */
/* tags 32995-32999 are private tags registered to SGI */
```

```
#define TIFFTAG_MATTEING        32995   /* $use ExtraSamples */
#define TIFFTAG_DATATYPE        32996   /* $use SampleFormat */
#define TIFFTAG_IMAGEDEPTH      32997   /* z depth of image */
#define TIFFTAG_TILEDEPTH       32998   /* z depth/data tile */
#endif /* _TIFF_ p */
```

### TIFF I/O Library Headers

Now we define structures for the TIFF I/O library. The definitions are separated into two files, tiffio.h, which contains the public definitions of use to programs that call the library, and tiffioP.h, which contains the private definitions used with the library itself. We start with the public definitions.

```
/* header tiffio.h */
#ifndef _TIFFIO_
#define _TIFFIO_

/*
 * TIFF I/O Library Definitions.
 */
#include "tiff.h"

/*
 * TIFF is defined as an incomplete type to hide the
 * library's internal data structures from clients.
 */
typedef struct tiff TIFF;

#ifndef NULL
#define NULL    0
#endif
```

The public function pointer TIFFErrorHandler lets the application catch TIFF errors by installing a pointer to its own error handler there. The default version prints an error report to the standard output.

```
#if defined(c_plusplus) || defined(__cplusplus) ||
    defined(__STDC__) || defined(__EXTENDED__) || USE_PROTOTYPES
#include <stdio.h>
#include <stdarg.h>

typedef void (*TIFFErrorHandler)(char* module, char* fmt,
              va_list ap);
```

These are declarations of all the public routines in the library.

```
#if defined(__cplusplus)
extern "C" {
#endif
```

```
extern  void TIFFClose(TIFF*);
extern  int TIFFFlush(TIFF*);
extern  int TIFFFlushData(TIFF*);
extern  int TIFFGetField(TIFF*, int, ...);
extern  int TIFFVGetField(TIFF*, int, va_list);
extern  int TIFFGetFieldDefaulted(TIFF*, int, ...);
extern  int TIFFVGetFieldDefaulted(TIFF*, int, va_list);
extern  int TIFFReadDirectory(TIFF*);
extern  int TIFFScanlineSize(TIFF*);
extern  unsigned long TIFFStripSize(TIFF*);
extern  unsigned long TIFFVStripSize(TIFF*, unsigned long);
extern  int TIFFFileno(TIFF*);
extern  int TIFFGetMode(TIFF*);
extern  int TIFFIsTiled(TIFF*);
extern  long TIFFCurrentRow(TIFF*);
extern  int TIFFCurrentDirectory(TIFF*);
extern  int TIFFCurrentStrip(TIFF*);
extern  int TIFFReadBufferSetup(TIFF*, char*, unsigned);
extern  int TIFFSetDirectory(TIFF*, int);
extern  int TIFFSetField(TIFF*, int, ...);
extern  int TIFFVSetField(TIFF*, int, va_list);
extern  int TIFFWriteDirectory(TIFF *);
#if defined(c_plusplus) || defined(__cplusplus)
extern  TIFF* TIFFOpen(const char*, const char*);
extern  TIFF* TIFFFdOpen(const int, const char*, const char*);
extern  const char* TIFFFileName(TIFF*);
extern  void TIFFError(const char*, const char*, ...);
extern  void TIFFWarning(const char*, const char*, ...);
extern  void TIFFPrintDirectory(TIFF*, FILE*, long = 0);
extern  int TIFFReadScanline(TIFF*, unsigned char*, unsigned,
            unsigned = 0);
extern  int TIFFWriteScanline(TIFF*, unsigned char*, unsigned,
            unsigned = 0);
extern  int TIFFReadRGBAImage(TIFF*, unsigned long,
            unsigned long, unsigned long*, int stop = 0);
#else
extern  TIFF* TIFFOpen(char*, char*);
extern  TIFF* TIFFFdOpen(int, char*, char*);
extern  char* TIFFFileName(TIFF*);
extern  void TIFFError(char*, char*, ...);
extern  TIFFErrorHandler TIFFSetErrorHandler(TIFFErrorHandler
                                             handler);
extern  void TIFFWarning(char*, char*, ...);
extern  TIFFErrorHandler TIFFSetWarningHandler(TIFFErrorHandler
                                               handler);
extern  void TIFFPrintDirectory(TIFF*, FILE*, long);
extern  int TIFFReadScanline(TIFF*, unsigned char*, unsigned,
                                                   unsigned);
extern  int TIFFWriteScanline(TIFF*, unsigned char*,
```

```
                                                    unsigned, unsigned);
extern  int TIFFReadRGBAImage(TIFF*, unsigned long,
                        unsigned long, unsigned long*, int stop);
#endif
extern  unsigned int TIFFComputeStrip(TIFF*, unsigned long,
                                                    unsigned int);
extern  unsigned int TIFFNumberOfStrips(TIFF*);
extern  int TIFFReadEncodedStrip(TIFF*, unsigned,
                                    unsigned char*, unsigned);
extern  int TIFFReadRawStrip(TIFF*, unsigned,
                                    unsigned char*, unsigned);
extern  int TIFFWriteEncodedStrip(TIFF*, unsigned,
                                    unsigned char*, unsigned);
extern  int TIFFWriteRawStrip(TIFF*, unsigned,
                                    unsigned char*, unsigned);
extern  int TIFFSwabShort(unsigned short *);
extern  int TIFFSwabLong(unsigned long *);
extern  int TIFFSwabArrayOfShort(unsigned short *, int);
extern  int TIFFSwabArrayOfLong(unsigned long *, int);
extern  int TIFFReverseBits(unsigned char *, int);
#if defined(__cplusplus)
}
#endif
#else
/* non-prototype versions */
#endif
#endif /* _TIFFIO_ p */
```

The next header is tiffioP.h, the private header library internal routines. It starts with the TIFFDirectory structure, a huge list of all the single-item fields that can occur in an IFD. The first field in the structure, td_fieldsset[ ], is treated as a set of bits marking which of the rest of the fields actually contain data.

```
/* tiffioP.h -- library internal data structures */
#ifndef _TIFFIOP_
#define _TIFFIOP_
/*
 * ''Library-private'' definitions.
 */
#include "tiffcompat.h"
#include "tiff.h"

/*
 * Internal format of a TIFF directory entry.
 */
typedef struct {
    u_long  td_fieldsset[2]; /* bit vector of fields that are set */

    u_long  td_imagewidth, td_imagelength, td_imagedepth;
```

```
    u_long  td_tilewidth, td_tilelength, td_tiledepth;
    u_short td_subfiletype;
    u_short td_bitspersample;
    u_short td_sampleformat;
    u_short td_compression;
    u_short td_photometric;
    u_short td_threshholding;
    u_short td_fillorder;
    u_short td_orientation;
    u_short td_samplesperpixel;
    u_short td_predictor;
    u_long  td_rowsperstrip;
    u_long  td_minsamplevalue, td_maxsamplevalue;   /* XXX */
    float   td_xresolution, td_yresolution;
    u_short td_resolutionunit;
    u_short td_planarconfig;
    float   td_xposition, td_yposition;
    u_long  td_group3options;
    u_long  td_group4options;
    u_short td_pagenumber[2];
    u_short td_matteing;
    u_short td_cleanfaxdata;
    u_short td_badfaxrun;
    u_long  td_badfaxlines;
    u_short *td_colormap[3];
    u_short td_halftonehints[2];
    char    *td_documentname;
    char    *td_artist;
    char    *td_datetime;
    char    *td_hostcomputer;
    char    *td_imagedescription;
    char    *td_make;
    char    *td_model;
    char    *td_software;
    char    *td_pagename;
    u_long  td_stripsperimage;
    u_long  td_nstrips;     /* size of offset & bytecount arrays */
    u_long  *td_stripoffset;
    u_long  *td_stripbytecount;
} TIFFDirectory;

/*
 * Following are field flags used to indicate fields that have
 * been set in a directory and to reference fields
 * when manipulating a directory.
 */

/* multi-entry fields */
#define FIELD_IMAGEDIMENSIONS    0
```

```
#define FIELD_TILEDIMENSIONS     1
#define FIELD_CELLDIMENSIONS     2       /* XXX */
#define FIELD_RESOLUTION         3
#define FIELD_POSITION           4
/* single-entry fields */
#define FIELD_SUBFILETYPE        5
#define FIELD_BITSPERSAMPLE      6
#define FIELD_COMPRESSION        7
#define FIELD_PHOTOMETRIC        8
#define FIELD_THRESHHOLDING      9
#define FIELD_FILLORDER         10
#define FIELD_DOCUMENTNAME      11
#define FIELD_IMAGEDESCRIPTION  12
#define FIELD_MAKE              13
#define FIELD_MODEL             14
#define FIELD_ORIENTATION       15
#define FIELD_SAMPLESPERPIXEL   16
#define FIELD_ROWSPERSTRIP      17
#define FIELD_MINSAMPLEVALUE    18
#define FIELD_MAXSAMPLEVALUE    19
#define FIELD_PLANARCONFIG      20
#define FIELD_PAGENAME          21
#define FIELD_GROUP3OPTIONS     22
#define FIELD_GROUP4OPTIONS     23
#define FIELD_RESOLUTIONUNIT    24
#define FIELD_PAGENUMBER        25
#define FIELD_STRIPBYTECOUNTS   26
#define FIELD_STRIPOFFSETS      27
#define FIELD_COLORMAP          28
#define FIELD_PREDICTOR         29
#define FIELD_ARTIST            30
#define FIELD_DATETIME          31
#define FIELD_HOSTCOMPUTER      32
#define FIELD_SOFTWARE          33
#define FIELD_MATTEING          34
#define FIELD_BADFAXLINES       35
#define FIELD_CLEANFAXDATA      36
#define FIELD_BADFAXRUN         37
#define FIELD_SAMPLEFORMAT      38
#define FIELD_SMINSAMPLEVALUE   39
#define FIELD_SMAXSAMPLEVALUE   40
#define FIELD_IMAGEDEPTH        41
#define FIELD_TILEDEPTH         42
#define FIELD_HALFTONEHINTS     43
#define FIELD_LAST              59
```

The macros TIFFExtractData( ) and TIFFInsertData( ) take care of aligning data correctly when it is short enough that it fits within the offset field of an IFD entry.

They are used primarily to extract and store the bits-per-pixel and compression type fields.

```
#define TIFFExtractData(tif, type, v) \
    ((tif)->tif_header.tiff_magic == TIFF_BIGENDIAN ? \
        ((v) >> (tif)->tif_typeshift[type]) & \
                (tif)->tif_typemask[type] : \
    (v) & (tif)->tif_typemask[type])
#define TIFFInsertData(tif, type, v) \
    ((tif)->tif_header.tiff_magic == TIFF_BIGENDIAN ? \
        ((v) & (tif)->tif_typemask[type]) << \
                (tif)->tif_typeshift[type] : \
    (v) & (tif)->tif_typemask[type])
```

An array of TIFFFieldInfo structures stores a constant table of information about all possible IFD fields. We discuss this in detail below.

```
typedef struct {
    u_short field_tag;       /* field's tag */
    short   field_readcount;   /* read count (-1 for unknown) */
    short   field_writecount;  /* write count (-1 for unknown) */
    TIFFDataType field_type;   /* type of associated data */
    u_short field_bit;       /* bit in fieldsset bit vector */
    u_short field_oktochange;
                        /* if true, can change while writing */
    char    *field_name;         /* ASCII name */
} TIFFFieldInfo;

#define FIELD_IGNORE    ((u_short)-1)
                                /* tags processed but ignored */

#define TIFF_ANY    TIFF_NOTYPE /* for field descriptor searching */
#define TIFF_VARIABLE   -1  /* marker for variable length tags */
#define TIFF_SPP    -2      /* marker for SamplesPerPixel tags */

extern  const TIFFFieldInfo tiffFieldInfo[];
                                /* table of field descriptors */
extern  const int tiffDataWidth[];
                                /* table of tag datatype widths */
```

These macros test and set the bits in td_fieldsset[ ] that indicate which fields contain useful data.

```
#define BITn(n)             (((unsigned)1L)<<((n)&0x1f))
#define BITFIELDn(tif, n)
            ((tif)->tif_dir.td_fieldsset[(n)/32])
#define TIFFFieldSet(tif, field)
            (BITFIELDn(tif, field) & BITn(field))
#define TIFFSetFieldBit(tif, field) (BITFIELDn(tif, field) |=
```

```
                BITn(field))
#define TIFFClrFieldBit(tif, field) (BITFIELDn(tif, field) &= \
                ~BITn(field))

#define FieldSet(fields, f)     (fields[(f)/32] & BITn(f))
#define ResetFieldBit(fields, f)    (fields[(f)/32] &= ~BITn(f))
```

At last we get to the definition of the "tiff" structure (which is also the TIFF datatype) that holds all of the information necessary to process a TIFF file. Most of the fields are either self-explanatory or will be described later with the code that manipulates them. Notice the list of function pointers; these are set to call routines appropriate to the type of image compression that the file uses. (This is a good example of object-oriented design in C, by the way.)

```
struct tiff {
    char    *tif_name;      /* name of open file */
    short   tif_fd;         /* open file descriptor */
    short   tif_mode;       /* open mode (O_*) */
    char    tif_fillorder; /* natural bit fill order for machine */
    char    tif_options;    /* compression-specific options */
    short   tif_flags;
#define TIFF_DIRTYHEADER    0x1
                            /* header must be written on close */
#define TIFF_DIRTYDIRECT    0x2
                            /* current directory must be written */
#define TIFF_BUFFERSETUP    0x4 /* data buffers setup */
#define TIFF_BEENWRITING    0x8 /* written 1+ scanlines to file */
#define TIFF_SWAB       0x10    /* byte swap file information */
#define TIFF_NOBITREV       0x20    /* inhibit bit reversal logic */
#define TIFF_MYBUFFER       0x40
                             /* my raw data buffer; free on close */
#define TIFF_ISTILED        0x80 /* file is tile, not strip-based */
#define TIFF_MAPPED     0x100   /* file is mapped into memory */
#define TIFF_POSTENCODE     0x200
                            /* need call to postencode routine */
    long    tif_diroff;     /* file offset of current directory */
    long    tif_nextdiroff; /* file offset of following directory */
    TIFFDirectory tif_dir;  /* internal rep of current directory */
    TIFFHeader tif_header;  /* file's header block */
    int const *tif_typeshift;   /* datatype shift counts */
    long const *tif_typemask;   /* datatype masks */
    long    tif_row;        /* current scanline */
    int tif_curdir;     /* current directory (index) */
    int tif_curstrip;       /* current strip for read/write */
    long    tif_curoff;     /* current offset for read/write */
/* tiling support */
    long    tif_col;        /* current column (offset by row too) */
    int     tif_curtile;        /* current tile for read/write */
    long    tif_tilesize;       /* # of bytes in a tile */
```

```
/* compression scheme hooks */
    int (*tif_predecode)();  /* pre row/strip/tile decoding */
    int (*tif_preencode)();  /* pre row/strip/tile encoding */
    int (*tif_postencode)(); /* post row/strip/tile encoding */
    int (*tif_decoderow)();  /* scanline decoding routine */
    int (*tif_encoderow)();  /* scanline encoding routine */
    int (*tif_decodestrip)();   /* strip decoding routine */
    int (*tif_encodestrip)();   /* strip encoding routine */
    int (*tif_decodetile)();    /* tile decoding routine */
    int (*tif_encodetile)();    /* tile encoding routine */
    int (*tif_close)();     /* cleanup-on-close routine */
    int (*tif_seek)();      /* position within a strip routine */
    int (*tif_cleanup)();   /* routine called to cleanup state */
    char    *tif_data;      /* compression scheme private data */
/* input/output buffering */
    int tif_scanlinesize;    /* # of bytes in a scanline */
    int tif_scanlineskew;    /* scanline skew for reading strips */
    char    *tif_rawdata;    /* raw data buffer */
    long    tif_rawdatasize; /* # of bytes in raw data buffer */
    char    *tif_rawcp;      /* current spot in raw buffer */
    long    tif_rawcc;       /* bytes unread from raw buffer */
};
```

Now we include tiffio.h to pick up the public definitions from above. The macro howmany($x$, $y$) computes how many fields are needed to contain $x$ subfields if a field can contain $y$ subfields and roundup( ) rounds up $x$ to the next multiple of $y$.

```
#include "tiffio.h"

/* NB: the u_int casts are to silence certain ANSI-C compilers */
#ifdef howmany
#undef howmany
#endif
#define howmany(x, y)   ((((u_int)(x))+(((u_int)(y))-1))/ \
                                                ((u_int)(y)))
#ifdef roundup
#undef roundup
#endif
#define roundup(x, y)   (howmany(x,y)*((u_int)(y)))

/* define internal routines */

#if defined(__cplusplus)
extern "C" {
#endif
extern  TIFFFieldInfo const *TIFFFindFieldInfo(u_short,
                                                TIFFDataType);
extern  TIFFFieldInfo const *TIFFFieldWithTag(u_short);
```

```
extern  int _TIFFgetfield(TIFFDirectory*, int, ...);
extern  int TIFFNoRowEncode(TIFF*, u_char*, int, u_int);
extern  int TIFFNoStripEncode(TIFF*, u_char*, int, u_int);
extern  int TIFFNoTileEncode(TIFF*, u_char*, int, u_int);
extern  int TIFFNoRowDecode(TIFF*, u_char*, int, u_int);
extern  int TIFFNoStripDecode(TIFF*, u_char*, int, u_int);
extern  int TIFFNoTileDecode(TIFF*, u_char*, int, u_int);
#if defined(__cplusplus)
}
#endif
#endif /* _TIFFIOP_ p */
```

The include file machdep.h takes care of declaring the byte order and native floating point format of the local machine. The full version of this file on the companion disk is much longer than this—here we show just the parts that apply to a 286 or 386 system.

```
/* machdep.h - machine dependent definitions */
#if defined(m386) || defined(M_I86) || defined(i386)
                  || defined(MSDOS)
#define BIGENDIAN   0
#define IEEEFP          /* IEEE floating point supported */
#endif /* m386 || M_I86 || i386 || MSDOS */

#ifdef IEEEFP
/* do byte swapping for files in the other order */
#define TIFFSwabArrayOfFloat(fp,n)
        TIFFSwabArrayOfLong((unsigned long *)fp,n)
#define TIFFSwabArrayOfDouble(dp,n)
        TIFFSwabArrayOfLong((unsigned long *)dp,2*n)
#endif /* IEEEFP */

#endif /* _MACHDEP_ p */
```

Even though nearly every C compiler purports to be fully ANSI compatible, there are still many small areas where details vary from one compiler to another.

The header file tiffcompat.h includes special-case declarations for all of the compilers to which the package has been ported. Most of the special cases are controlled by symbols defined on the compiler's command line, notably:

**USE_CONST** Compiler supports the const declaration.

**USE_VARARGS** Use the older varargs.h header rather than the newer stdargs.h.

**USE_PROTOTYPES** Use ANSI prototypes rather than older K&R declarations.

**HAVE_IEEEFP** The native floating point is IEEE-compatible, so no conversion routines are needed.

**BSDTYPES** The system doesn't automatically define the BSD Unix type names such as u_char for unsigned char, so we have to do it ourselves.

**SYSV** UNIX System V compiler libraries available.

**MSDOS** MS-DOS libraries available.

```
/* tiffcompat.h - handle warts of various C compilers */
#ifndef _COMPAT_
#define _COMPAT_
/*
 * This file contains a hodgepodge of definitions and
 * declarations that are needed to provide compatibility
 * between the native system and the base UNIX implementation
 * that the library assumes (~4BSD).  In particular, you
 * can override the standard i/o interface (read/write/lseek)
 * by redefining the ReadOK/WriteOK/SeekOK macros to your
 * liking.
 *
 * NB: This file is a mess.
 */
#if (defined(__STDC__) || defined(__EXTENDED__)) \
     && !defined(USE_PROTOTYPES)
#define USE_PROTOTYPES  1
#define USE_CONST   1
#endif

#if !USE_CONST && !defined(const)
#define const
#endif

#ifdef THINK_C
#include <unix.h>
#include <math.h>
#endif
#if USE_PROTOTYPES
#include <stdio.h>
#endif
#ifndef THINK_C
#include <sys/types.h>
#endif
#ifdef VMS
#include <file.h>
#include <unixio.h>
#else
#include <fcntl.h>
#endif
#if defined(THINK_C) || defined(applec)
#include <stdlib.h>
#endif
```

```
 /* Workarounds for BSD lseek definitions.
 */
#if defined(SYSV) || defined(VMS) || defined(MSDOS)
#if defined(SYSV)
#include <unistd.h>
#endif
#define L_SET   SEEK_SET
#define L_INCR  SEEK_CUR
#define L_XTND  SEEK_END
#endif
#ifndef L_SET
#define L_SET   0
#define L_INCR  1
#define L_XTND  2
#endif

/*
 * SVID workarounds for BSD bit
 * string manipulation routines.
 */
#if defined(SYSV) || defined(THINK_C) || defined(applec) \
                  || defined(VMS) || defined(MSDOS)
#define bzero(dst,len)      memset((char *)dst, 0, len)
#define bcopy(src,dst,len)  memcpy((char *)dst, (char *)src, len)
#define bcmp(src, dst, len) memcmp((char *)dst, (char *)src, len)
#endif

/*
 * The BSD typedefs are used throughout the library.
 * If your system doesn't have them in <sys/types.h>,
 * then define BSDTYPES in your Makefile.
 */
#ifdef BSDTYPES
typedef unsigned char u_char;
typedef unsigned short u_short;
typedef unsigned int u_int;
typedef unsigned long u_long;
#endif

/*
 * Return an open file descriptor or -1.
 */
#if defined(applec) || defined(THINK_C)
#define TIFFOpenFile(name, mode, prot)  open(name, mode)
#else
#if defined(MSDOS)
#define TIFFOpenFile(name, mode, prot)  open(name,
        mode|O_BINARY, prot) \
```

```
#else
#define TIFFOpenFile(name, mode, prot)  open(name, mode, prot)
#endif
#endif

/*
 * Return the size in bytes of the file
 * associated with the supplied file descriptor.
 */
#if USE_PROTOTYPES
extern  long TIFFGetFileSize(int fd);
#else
extern  long TIFFGetFileSize();
#endif

/*
 * Mac workaround to handle the file
 * extension semantics of lseek.
 */
#ifdef applec
#define lseek   mpw_lseek
extern long mpw_lseek(int, long, int);
#else
extern  long lseek();
#endif

/*
 * Default Read/Seek/Write definitions.
 */
#ifndef ReadOK
#define ReadOK(fd, buf, size)  (read(fd, (char *)buf, size) == size)
#endif
#ifndef SeekOK
#define SeekOK(fd, off) (lseek(fd, (long)off, L_SET) == (long)off)
#endif
#ifndef WriteOK
#define WriteOK(fd, buf, size)  (write(fd, (char *)buf, size) \
                                                    == size)
#endif

#if defined(__MACH__) || defined(THINK_C)
extern  void *malloc(size_t size);
extern  void *realloc(void *ptr, size_t size);
#else /* !__MACH__ p && !THINK_C */
#if defined(MSDOS)
#include <malloc.h>
#else /* !MSDOS */
extern  char *malloc();
extern  char *realloc();
```

```
#endif /* _IBMR2 */
#endif /* !MSDOS */
#endif /* !__MACH__ p */

/*
 * dblparam_t is the type that a double precision
 * floating point value will have on the parameter
 * stack (when coerced by the compiler).
 */
#ifdef applec
typedef extended dblparam_t;
#else
typedef double dblparam_t;
#endif

/*
 * Varargs parameter list handling...YECH!!!!
 */
#if defined(__STDC__) && !defined(USE_VARARGS)
#define USE_VARARGS 0
#endif

#if defined(USE_VARARGS)
#if USE_VARARGS
#include <varargs.h>
#define VA_START(ap, parmN) va_start(ap)
#else
#include <stdarg.h>
#define VA_START(ap, parmN) va_start(ap, parmN)
#endif
#endif /* defined(USE_VARARGS) */
#endif /* _COMPAT_ p */
```

The final include file, prototypes.h, defines some macros that allow the same code to compile on modern C compilers that have ANSI prototypes and on traditional C compilers that don't. The macro DECLARE3( ), for example, is used to declare a function with three arguments. The macro itself takes as its arguments the name of the function and the types and names of the three arguments.

```
/* prototypes.h - macros to handle prototypes or the lack thereof */

#if USE_PROTOTYPES
#define DECLARE1(f,t1,a1)          f(t1 a1)
#define DECLARE2(f,t1,a1,t2,a2)      f(t1 a1, t2 a2)
#define DECLARE3(f,t1,a1,t2,a2,t3,a3)   f(t1 a1, t2 a2, t3 a3)
#define DECLARE4(f,t1,a1,t2,a2,t3,a3,t4,a4)\
    f(t1 a1, t2 a2, t3 a3, t4 a4)
#define DECLARE5(f,t1,a1,t2,a2,t3,a3,t4,a4,t5,a5)\
    f(t1 a1, t2 a2, t3 a3, t4 a4, t5 a5)
```

```
#define DECLARE6(f,t1,a1,t2,a2,t3,a3,t4,a4,t5,a5,t6,a6)\
    f(t1 a1, t2 a2, t3 a3, t4 a4, t5 a5, t6 a6)

/* varargs routines */
#define DECLARE1V(f,t1,a1)      f(t1 a1 ...)
#define DECLARE2V(f,t1,a1,t2,a2)    f(t1 a1, t2 a2, ...)
#define DECLARE3V(f,t1,a1,t2,a2,t3,a3)  f(t1 a1, t2 a2, t3 a3, ...)
#else   /* no prototypes */
#define DECLARE1(f,t1,a1)       f(a1) t1 a1;
#define DECLARE2(f,t1,a1,t2,a2)     f(a1,a2) t1 a1; t2 a2;
#define DECLARE3(f,t1,a1,t2,a2,t3,a3)   f(a1, a2, a3) t1 a1; \
                t2 a2; t3 a3;
#define DECLARE4(f,t1,a1,t2,a2,t3,a3,t4,a4) \
    f(a1, a2, a3, a4) t1 a1; t2 a2; t3 a3; t4 a4;
#define DECLARE5(f,t1,a1,t2,a2,t3,a3,t4,a4,t5,a5)\
    f(a1, a2, a3, a4, a5) t1 a1; t2 a2; t3 a3; t4 a4; t5 a5;
#define DECLARE6(f,t1,a1,t2,a2,t3,a3,t4,a4,t5,a5,t6,a6)\
    f(a1, a2, a3, a4, a5, a6) t1 a1; t2 a2; t3 a3; t4 a4; \
                                t5 a5; t6 a6;

#if USE_VARARGS
#define DECLARE1V(f,t1,a1) \
    f(a1, va_alist) t1 a1; va_dcl
#define DECLARE2V(f,t1,a1,t2,a2) \
    f(a1, a2, va_alist) t1 a1; t2 a2; va_dcl
#define DECLARE3V(f,t1,a1,t2,a2,t3,a3) \
    f(a1, a2, a3, va_alist) t1 a1; t2 a2; t3 a3; va_dcl
#else
"Help, I don't know how to handle this case:
                                !USE_PROTOTYPES and !USE_VARARGS?"
#endif
#endif
```

## Handling IFDs

The heart of any TIFF file is the IFDs, so the first code we look at manages the IFDs. Since all of the IFD fields have slightly different rules, semantics, and interpretations, we need a list of the valid field types, which we find in the array tiffFieldInfo.

Each array entry consists of:

**field_tag** The tag value in a TIFF file.

**field_readcount** The number of items this field should contain when reading a file, or −1 for variable.

**field_writecount** The number of items this field should contain when reading a file, or −1 for variable. (This is different from the read count, because

there are a few fields where we read variable numbers of items but write only one.)

**field_type** The datatype of the field.

**field_bit** The bit in an in-memory TIFFDirectory that corresponds to this field.

**field_oktochange** Non-zero if this field can be changed halfway through writing a file.

**field_name** The field name as a string, for debugging.

```
/* tif_dirinfo.c
 * TIFF Library.
 *
 * Known Directory Tag Support.
 */
#include "tiffioP.h"
#include "prototypes.h"

#ifndef TRUE
#define TRUE    1
#define FALSE   0
#endif

/*
 * NB: THIS ARRAY IS ASSUMED TO BE SORTED BY TAG.
 *     Also, if a tag can have both LONG and SHORT types
 *     then the LONG must be placed before the SHORT for
 *     writing to work properly.
 */
const TIFFFieldInfo tiffFieldInfo[] = {
    { TIFFTAG_SUBFILETYPE,    1, 1, TIFF_LONG,   FIELD_SUBFILETYPE,
      TRUE, "SubfileType" },
/* XXX SHORT for compatibility w/ old versions of the library */
    { TIFFTAG_SUBFILETYPE,    1, 1, TIFF_SHORT,  FIELD_SUBFILETYPE,
      TRUE, "SubfileType" },
    { TIFFTAG_OSUBFILETYPE,   1, 1, TIFF_SHORT,  FIELD_SUBFILETYPE,
      TRUE, "OldSubfileType" },
    { TIFFTAG_IMAGEWIDTH,     1, 1, TIFF_LONG,
                                          FIELD_IMAGEDIMENSIONS,
      FALSE,    "ImageWidth" },
    { TIFFTAG_IMAGEWIDTH,     1, 1, TIFF_SHORT,
                                          FIELD_IMAGEDIMENSIONS,
      FALSE,    "ImageWidth" },
    { TIFFTAG_IMAGELENGTH,    1, 1, TIFF_LONG,
                                          FIELD_IMAGEDIMENSIONS,
      TRUE, "ImageLength" },
    { TIFFTAG_IMAGELENGTH,    1, 1, TIFF_SHORT,
```

```
                                                    FIELD_IMAGEDIMENSIONS,
  TRUE, "ImageLength" },
{ TIFFTAG_BITSPERSAMPLE,    -1,-1, TIFF_SHORT,
                                                    FIELD_BITSPERSAMPLE,
  FALSE,    "BitsPerSample" },
{ TIFFTAG_COMPRESSION,  -1, 1, TIFF_SHORT,  FIELD_COMPRESSION,
  FALSE,    "Compression" },
{ TIFFTAG_PHOTOMETRIC,   1, 1, TIFF_SHORT,  FIELD_PHOTOMETRIC,
  FALSE,    "PhotometricInterpretation" },
{ TIFFTAG_THRESHHOLDING,     1, 1, TIFF_SHORT,
                                                    FIELD_THRESHHOLDING,
  TRUE, "Threshholding" },
{ TIFFTAG_CELLWIDTH,     1, 1, TIFF_SHORT,  FIELD_IGNORE,
  TRUE, "CellWidth" },
{ TIFFTAG_CELLLENGTH,    1, 1, TIFF_SHORT,  FIELD_IGNORE,
  TRUE, "CellLength" },
{ TIFFTAG_FILLORDER,     1, 1, TIFF_SHORT,  FIELD_FILLORDER,
  FALSE,    "FillOrder" },
{ TIFFTAG_DOCUMENTNAME, -1,-1, TIFF_ASCII,  FIELD_DOCUMENTNAME,
  TRUE, "DocumentName" },
{ TIFFTAG_IMAGEDESCRIPTION, -1,-1, TIFF_ASCII,
                                                    FIELD_IMAGEDESCRIPTION,
  TRUE, "ImageDescription" },
{ TIFFTAG_MAKE,     -1,-1, TIFF_ASCII,  FIELD_MAKE,
  TRUE, "Make" },
{ TIFFTAG_MODEL,        -1,-1, TIFF_ASCII,  FIELD_MODEL,
  TRUE, "Model" },
{ TIFFTAG_STRIPOFFSETS, -1,-1, TIFF_LONG,   FIELD_STRIPOFFSETS,
  FALSE,    "StripOffsets" },
{ TIFFTAG_STRIPOFFSETS, -1,-1, TIFF_SHORT,  FIELD_STRIPOFFSETS,
  FALSE,    "StripOffsets" },
{ TIFFTAG_ORIENTATION,   1, 1, TIFF_SHORT,  FIELD_ORIENTATION,
  FALSE,    "Orientation" },
{ TIFFTAG_SAMPLESPERPIXEL,   1, 1, TIFF_SHORT,
                                                    FIELD_SAMPLESPERPIXEL,
  FALSE,    "SamplesPerPixel" },
{ TIFFTAG_ROWSPERSTRIP,  1, 1, TIFF_LONG,   FIELD_ROWSPERSTRIP,
  FALSE,    "RowsPerStrip" },
{ TIFFTAG_ROWSPERSTRIP,  1, 1, TIFF_SHORT,  FIELD_ROWSPERSTRIP,
  FALSE,    "RowsPerStrip" },
{ TIFFTAG_STRIPBYTECOUNTS,  -1,-1, TIFF_LONG,
                                                    FIELD_STRIPBYTECOUNTS,
  FALSE,    "StripByteCounts" },
{ TIFFTAG_STRIPBYTECOUNTS,  -1,-1, TIFF_SHORT,
                                                    FIELD_STRIPBYTECOUNTS,
  FALSE,    "StripByteCounts" },
{ TIFFTAG_MINSAMPLEVALUE,   -2,-1, TIFF_SHORT,
                                                    FIELD_MINSAMPLEVALUE,
  TRUE, "MinSampleValue" },
```

```
{ TIFFTAG_MAXSAMPLEVALUE,   -2,-1, TIFF_SHORT,
                                        FIELD_MAXSAMPLEVALUE,
  TRUE, "MaxSampleValue" },
{ TIFFTAG_XRESOLUTION,   1, 1, TIFF_RATIONAL, FIELD_RESOLUTION,
  FALSE,    "XResolution" },
{ TIFFTAG_YRESOLUTION,   1, 1, TIFF_RATIONAL, FIELD_RESOLUTION,
  FALSE,    "YResolution" },
{ TIFFTAG_PLANARCONFIG,  1, 1, TIFF_SHORT,  FIELD_PLANARCONFIG,
  FALSE,    "PlanarConfiguration" },
{ TIFFTAG_PAGENAME,     -1,-1, TIFF_ASCII,  FIELD_PAGENAME,
  TRUE, "PageName" },
{ TIFFTAG_XPOSITION,     1, 1, TIFF_RATIONAL,   FIELD_POSITION,
  TRUE, "XPosition" },
{ TIFFTAG_YPOSITION,     1, 1, TIFF_RATIONAL,   FIELD_POSITION,
  TRUE, "YPosition" },
{ TIFFTAG_FREEOFFSETS,  -1,-1, TIFF_LONG,   FIELD_IGNORE,
  FALSE,    "FreeOffsets" },
{ TIFFTAG_FREEBYTECOUNTS,   -1,-1, TIFF_LONG,   FIELD_IGNORE,
  FALSE,    "FreeByteCounts" },
{ TIFFTAG_GRAYRESPONSEUNIT,  1, 1, TIFF_SHORT,  FIELD_IGNORE,
  TRUE, "GrayResponseUnit" },
{ TIFFTAG_GRAYRESPONSECURVE,-1,-1, TIFF_SHORT,  FIELD_IGNORE,
  TRUE, "GrayResponseCurve" },
{ TIFFTAG_GROUP3OPTIONS,     1, 1, TIFF_LONG,
                                        FIELD_GROUP3OPTIONS,
  FALSE,    "Group3Options" },
{ TIFFTAG_GROUP4OPTIONS,     1, 1, TIFF_LONG,
                                        FIELD_GROUP4OPTIONS,
  FALSE,    "Group4Options" },
{ TIFFTAG_RESOLUTIONUNIT,    1, 1, TIFF_SHORT,
                                      FIELD_RESOLUTIONUNIT,
  FALSE,    "ResolutionUnit" },
{ TIFFTAG_PAGENUMBER,    2, 2, TIFF_SHORT,  FIELD_PAGENUMBER,
  TRUE, "PageNumber" },
{ TIFFTAG_COLORRESPONSEUNIT, 1, 1, TIFF_SHORT,  FIELD_IGNORE,
  TRUE, "ColorResponseUnit" },
{ TIFFTAG_SOFTWARE,     -1,-1, TIFF_ASCII,  FIELD_SOFTWARE,
  TRUE, "Software" },
{ TIFFTAG_DATETIME,     20,20, TIFF_ASCII,  FIELD_DATETIME,
  TRUE, "DateTime" },
{ TIFFTAG_ARTIST,       -1,-1, TIFF_ASCII,  FIELD_ARTIST,
  TRUE, "Artist" },
{ TIFFTAG_HOSTCOMPUTER, -1,-1, TIFF_ASCII,  FIELD_HOSTCOMPUTER,
  TRUE, "HostComputer" },
{ TIFFTAG_PREDICTOR,     1, 1, TIFF_SHORT,  FIELD_PREDICTOR,
  FALSE,    "Predictor" },
{ TIFFTAG_COLORMAP,     -1,-1, TIFF_SHORT,  FIELD_COLORMAP,
  TRUE, "ColorMap" },
{ TIFFTAG_HALFTONEHINTS,     2, 2, TIFF_SHORT,
```

```
                                                    FIELD_HALFTONEHINTS,
    TRUE, "HalftoneHints" },
{ TIFFTAG_TILEWIDTH,      1, 1, TIFF_LONG,  FIELD_TILEDIMENSIONS,
    FALSE,    "TileWidth" },
{ TIFFTAG_TILEWIDTH,      1, 1, TIFF_SHORT, FIELD_TILEDIMENSIONS,
    FALSE,    "TileWidth" },
{ TIFFTAG_TILELENGTH,     1, 1, TIFF_LONG,  FIELD_TILEDIMENSIONS,
    FALSE,    "TileLength" },
{ TIFFTAG_TILELENGTH,     1, 1, TIFF_SHORT, FIELD_TILEDIMENSIONS,
    FALSE,    "TileLength" },
{ TIFFTAG_TILEOFFSETS,   -1, 1, TIFF_LONG,  FIELD_STRIPOFFSETS,
    FALSE,    "TileOffsets" },
{ TIFFTAG_TILEBYTECOUNTS,    -1, 1, TIFF_LONG,
                                                FIELD_STRIPBYTECOUNTS,
    FALSE,    "TileByteCounts" },
{ TIFFTAG_TILEBYTECOUNTS,    -1, 1, TIFF_SHORT,
                                                FIELD_STRIPBYTECOUNTS,
    FALSE,    "TileByteCounts" },
{ TIFFTAG_BADFAXLINES,    1, 1, TIFF_LONG,    FIELD_BADFAXLINES,
    TRUE, "BadFaxLines" },
{ TIFFTAG_BADFAXLINES,    1, 1, TIFF_SHORT,   FIELD_BADFAXLINES,
    TRUE, "BadFaxLines" },
{ TIFFTAG_CLEANFAXDATA,   1, 1, TIFF_SHORT,   FIELD_CLEANFAXDATA,
    TRUE, "CleanFaxData" },
{ TIFFTAG_CONSECUTIVEBADFAXLINES,1,1,
                                              TIFF_LONG,FIELD_BADFAXRUN,
    TRUE, "ConsecutiveBadFaxLines" },
{ TIFFTAG_CONSECUTIVEBADFAXLINES,1,1,
                                             TIFF_SHORT,FIELD_BADFAXRUN,
    TRUE, "ConsecutiveBadFaxLines" },
{ TIFFTAG_EXTRASAMPLES, -1,-1, TIFF_SHORT,   FIELD_MATTEING,
    FALSE,    "ExtraSamples" },
{ TIFFTAG_SAMPLEFORMAT, -1,-1, TIFF_SHORT,   FIELD_SAMPLEFORMAT,
    FALSE,    "SampleFormat" },
    { 0 }
};
```

The array tiffDataWidth[ ], indexed by field types, gives the size in bytes of that type.

```
const int tiffDataWidth[] = {
    1,  /* nothing */
    1,  /* TIFF_BYTE */
    1,  /* TIFF_ASCII */
    2,  /* TIFF_SHORT */
    4,  /* TIFF_LONG */
    8,  /* TIFF_RATIONAL */
    1,  /* TIFF_SBYTE */
    1,  /* TIFF_UNDEFINED */
```

```
    2,  /* TIFF_SSHORT */
    4,  /* TIFF_SLONG */
    8,  /* TIFF_SRATIONAL */
    4,  /* TIFF_FLOAT */
    8,  /* TIFF_DOUBLE */
};
```

A few routines help us use the table. TIFFFindFieldInfo( ) takes two arguments, the tag and the datatype, and returns a pointer to the appropriate field. The datatype argument lets us handle a few of the fields that can have either short or long values. TIFFFieldWithTag( ) is a wrapper around that routine that finds a field regardless of datatype.

```
/* get entry with tag, type dt, or any type if dt is TIFF_ANY */
TIFFFieldInfo const *
DECLARE2(TIFFFindFieldInfo, u_short, tag, TIFFDataType, dt)
{
    static TIFFFieldInfo const *last = NULL;
    register TIFFFieldInfo const *fip;

    if (last && last->field_tag == tag &&
        (dt == TIFF_ANY || dt == last->field_type))
        return (last);
    /* NB: if table gets big, use sorted search
                                        (e.g., binary search) */
    for (fip = tiffFieldInfo; fip->field_tag; fip++)
        if (fip->field_tag == tag &&
            (dt == TIFF_ANY || fip->field_type == dt))
            return (last = fip);
    return ((TIFFFieldInfo *)0);
}

TIFFFieldInfo const *
DECLARE1(TIFFFieldWithTag, u_short, tag)
{
    TIFFFieldInfo const *fip = TIFFFindFieldInfo(tag, TIFF_ANY);
    if (fip)
        return (fip);
    TIFFError("TIFFFieldWithTag",
              "Internal error, unknown tag 0x%x", tag);
    exit(-1);
    /*NOTREACHED*/
}
```

In tif_dir.c are the central routines that manipulate the in-memory TIFF directory information. First come utility routines used to copy field data of various types into data fields in the TIFFDirectory.

```
/* tif_dir.c
```

```
 * TIFF Library.
 *
 * Directory Tag Get & Set Routines.
 * (and also some miscellaneous stuff)
 *
 * NB: Beware of the varargs declarations for routines in
 *     this file.  The names and types of variables have been
 *     carefully chosen to make things work with compilers that
 *     are busted in one way or another (e.g., SGI/MIPS).
 */
#include "tiffioP.h"
#include "prototypes.h"

/* replace *cpp with a copy of string cp */
static
DECLARE2(setString, char**, cpp, char*, cp)
{
    if (*cpp)
        free(*cpp), *cpp = 0;
    if (cp) {
        int len = strlen(cp)+1;
        if (*cpp = malloc(len))
            bcopy(cp, *cpp, len);
    }
}

/* replace *wpp with a copy of array wp[n] */
static
DECLARE3(setShortArray, u_short**, wpp, u_short*, wp, long, n)
{
    if (*wpp)
        free((char *)*wpp), *wpp = 0;
    n *= sizeof (u_short);
    if (wp && (*wpp = (u_short *)malloc(n)))
        bcopy(wp, *wpp, n);
}

/* replace *wpp with a copy of array wp[n] */
static
DECLARE3(setLongArray, u_long**, wpp, u_long*, wp, long, n)
{
    if (*wpp)
        free((char *)*wpp), *wpp = 0;
    n *= sizeof (u_long);
    if (wp && (*wpp = (u_long *)malloc(n)))
        bcopy(wp, *wpp, n);
}

/* replace *wpp with a copy of array wp[n] */
```

```
static
DECLARE3(setFloatArray, float**, wpp, float*, wp, long, n)
{
    if (*wpp)
        free((char *)*wpp), *wpp = 0;
    n *= sizeof (float);
    if (wp && (*wpp = (float *)malloc(n)))
        bcopy(wp, *wpp, n);
}
```

TIFFSetField1( ) is the basic routine that sets any of the fields in a TIFF directory. It consists mostly of a big switch with entries that fetch an appropriate argument type from the list and store it in the directory.

```
/* In directory tif, set field with tag appropriately */
static
TIFFSetField1(tif, tag, ap)
    TIFF *tif;
    int tag;
    va_list ap;
{
    TIFFDirectory *td = &tif->tif_dir;
    int i, status = 1;
    long v;

    switch (tag) {
    case TIFFTAG_SUBFILETYPE:
        td->td_subfiletype = va_arg(ap, u_long);
        break;
    case TIFFTAG_IMAGEWIDTH:
        td->td_imagewidth = va_arg(ap, u_long);
        break;
    case TIFFTAG_IMAGELENGTH:
        td->td_imagelength = va_arg(ap, u_long);
        break;
    case TIFFTAG_BITSPERSAMPLE:
        td->td_bitspersample = va_arg(ap, int);
        break;
    case TIFFTAG_COMPRESSION:
        v = va_arg(ap, int) & 0xffff;
        /*
         * If we're changing the compression scheme,
         * then notify the previous module so that it
         * can clean up any state it's set up.
         */
        if (TIFFFieldSet(tif, FIELD_COMPRESSION)) {
            if (td->td_compression == v)
                break;
            if (tif->tif_cleanup)
```

```
                (*tif->tif_cleanup)(tif);
        }
        /*
         * Set up new compression routine state.
         */
        if (status = TIFFSetCompressionScheme(tif, v))
            td->td_compression = v;
        break;
    case TIFFTAG_PHOTOMETRIC:
        td->td_photometric = va_arg(ap, int);
        break;
    case TIFFTAG_THRESHHOLDING:
        td->td_threshholding = va_arg(ap, int);
        break;
    case TIFFTAG_FILLORDER:
        v = va_arg(ap, int);
        if (v != FILLORDER_LSB2MSB && v != FILLORDER_MSB2LSB)
            goto badvalue;
        td->td_fillorder = v;
        break;
    case TIFFTAG_DOCUMENTNAME:
        setString(&td->td_documentname, va_arg(ap, char *));
        break;
    case TIFFTAG_ARTIST:
        setString(&td->td_artist, va_arg(ap, char *));
        break;
    case TIFFTAG_DATETIME:
        setString(&td->td_datetime, va_arg(ap, char *));
        break;
    case TIFFTAG_HOSTCOMPUTER:
        setString(&td->td_hostcomputer, va_arg(ap, char *));
        break;
    case TIFFTAG_IMAGEDESCRIPTION:
        setString(&td->td_imagedescription, va_arg(ap, char *));
        break;
    case TIFFTAG_MAKE:
        setString(&td->td_make, va_arg(ap, char *));
        break;
    case TIFFTAG_MODEL:
        setString(&td->td_model, va_arg(ap, char *));
        break;
    case TIFFTAG_SOFTWARE:
        setString(&td->td_software, va_arg(ap, char *));
        break;
    case TIFFTAG_ORIENTATION:
        v = va_arg(ap, int);
        if (v < ORIENTATION_TOPLEFT || ORIENTATION_LEFTBOT < v) {
            TIFFWarning(tif->tif_name,
                "Bad value %ld for \"%s\" tag ignored",
```

```
                    v, TIFFFieldWithTag(tag)->field_name);
            } else
                td->td_orientation = v;
            break;
        case TIFFTAG_SAMPLESPERPIXEL:
            /* XXX should cross-check -- e.g., if palette, then 1 */
            v = va_arg(ap, int);
            if (v == 0)
                goto badvalue;
            if (v > 4) {
                TIFFError(tif->tif_name,
                    "Cannot handle %ld-channel data", v);
                status = 0;
            } else
                td->td_samplesperpixel = v;
            break;
        case TIFFTAG_ROWSPERSTRIP:
            v = va_arg(ap, u_long);
            if (v == 0)
                goto badvalue;
            td->td_rowsperstrip = v;
            if (!TIFFFieldSet(tif, FIELD_TILEDIMENSIONS)) {
                td->td_tilelength = v;
                td->td_tilewidth = td->td_imagewidth;
            }
            break;
        case TIFFTAG_MINSAMPLEVALUE:
            td->td_minsamplevalue = va_arg(ap, int) & 0xffff;
            break;
        case TIFFTAG_MAXSAMPLEVALUE:
            td->td_maxsamplevalue = va_arg(ap, int) & 0xffff;
            break;
        case TIFFTAG_XRESOLUTION:
            td->td_xresolution = va_arg(ap, dblparam_t);
            break;
        case TIFFTAG_YRESOLUTION:
            td->td_yresolution = va_arg(ap, dblparam_t);
            break;
        case TIFFTAG_PLANARCONFIG:
            v = va_arg(ap, int);
            if (v != PLANARCONFIG_CONTIG && v != PLANARCONFIG_SEPARATE)
                goto badvalue;
            td->td_planarconfig = v;
            break;
        case TIFFTAG_PAGENAME:
            setString(&td->td_pagename, va_arg(ap, char *));
            break;
        case TIFFTAG_XPOSITION:
            td->td_xposition = va_arg(ap, dblparam_t);
```

```
            break;
        case TIFFTAG_YPOSITION:
            td->td_yposition = va_arg(ap, dblparam_t);
            break;
        case TIFFTAG_GROUP3OPTIONS:
            td->td_group3options = va_arg(ap, u_long);
            break;
        case TIFFTAG_GROUP4OPTIONS:
            td->td_group4options = va_arg(ap, u_long);
            break;
        case TIFFTAG_RESOLUTIONUNIT:
            v = va_arg(ap, int);
            if (v < RESUNIT_NONE || RESUNIT_CENTIMETER < v)
                goto badvalue;
            td->td_resolutionunit = v;
            break;
        case TIFFTAG_PAGENUMBER:
            td->td_pagenumber[0] = va_arg(ap, int);
            td->td_pagenumber[1] = va_arg(ap, int);
            break;
        case TIFFTAG_HALFTONEHINTS:
            td->td_halftonehints[0] = va_arg(ap, int);
            td->td_halftonehints[1] = va_arg(ap, int);
            break;
        case TIFFTAG_COLORMAP:
            v = 1L<<td->td_bitspersample;
            setShortArray(&td->td_colormap[0], va_arg(ap, u_short *),
                          v);
            setShortArray(&td->td_colormap[1], va_arg(ap, u_short *),
                          v);
            setShortArray(&td->td_colormap[2], va_arg(ap, u_short *),
                          v);
            break;
        case TIFFTAG_PREDICTOR:
            td->td_predictor = va_arg(ap, int);
            break;
        case TIFFTAG_EXTRASAMPLES:
            v = va_arg(ap, int);
            if (v > td->td_samplesperpixel)
                goto badvalue;
            if (v != 1)             /* XXX */
                goto badvalue;      /* XXX */
            v = va_arg(ap, int);
            if (v != EXTRASAMPLE_ASSOCALPHA)/* XXX */
                goto badvalue;      /* XXX */
            td->td_matteing = 1;
            break;
        case TIFFTAG_MATTEING:
            td->td_matteing = va_arg(ap, int);
```

```
            break;
        case TIFFTAG_BADFAXLINES:
            td->td_badfaxlines = va_arg(ap, u_long);
            break;
        case TIFFTAG_CLEANFAXDATA:
            td->td_cleanfaxdata = va_arg(ap, int);
            break;
        case TIFFTAG_CONSECUTIVEBADFAXLINES:
            td->td_badfaxrun = va_arg(ap, u_long);
            break;
        case TIFFTAG_TILEWIDTH:
            v = va_arg(ap, u_long);
            if (v % 8)
                goto badvalue;
            td->td_tilewidth = v;
            tif->tif_flags |= TIFF_ISTILED;
            break;
        case TIFFTAG_TILELENGTH:
            v = va_arg(ap, u_long);
            if (v % 8)
                goto badvalue;
            td->td_tilelength = v;
            tif->tif_flags |= TIFF_ISTILED;
            break;
        case TIFFTAG_TILEDEPTH:
            v = va_arg(ap, u_long);
            if (v == 0)
                goto badvalue;
            td->td_tiledepth = v;
            break;
        case TIFFTAG_DATATYPE:
        case TIFFTAG_SAMPLEFORMAT:
            v = va_arg(ap, int);
            if (tag == TIFFTAG_DATATYPE && v == 0)
                v = SAMPLEFORMAT_VOID;
            if (v < SAMPLEFORMAT_INT || SAMPLEFORMAT_VOID < v)
                goto badvalue;
            td->td_sampleformat = v;
            break;
        case TIFFTAG_IMAGEDEPTH:
            td->td_imagedepth = va_arg(ap, u_long);
            break;
        default:
            TIFFError(tif->tif_name,
                "Internal error, tag value botch, tag \"%s\"",
                TIFFFieldWithTag(tag)->field_name);
            status = 0;
            break;
        }
```

Unless the change failed, also turn on the appropriate field bit to note that this field is set. Attempts to set a valid field with a value not appropriate for that field fall to the label badvalue, which prints a complaint.

```
    if (status) {
        TIFFSetFieldBit(tif, TIFFFieldWithTag(tag)->field_bit);
        tif->tif_flags |= TIFF_DIRTYDIRECT;
    }
    va_end(ap);
    return (status);
badvalue:
    TIFFError(tif->tif_name, "%ld: Bad value for \"%s\"", v,
        TIFFFieldWithTag(tag)->field_name);
    va_end(ap);
    return (0);
}
```

Many tags can only be changed before any image data is written to the file. The auxiliary OkToChangeTag( ) ensures that it is still legitimate to change a particular tag, either because no image has been written, or because the tag can still be changed. The public routines TIFFSetField( ) and TIFFVSetField( ) validate the proposed change using OkToChangeTag( ), and then use TIFFSetField1( ) to make the same change.

```
/*
 * Return 1/0 according to whether or not
 * it is permissible to set the tag's value.
 * Note that we allow ImageLength to be changed
 * so that we can append and extend to images.
 * Any other tag may not be altered once writing
 * has commenced, unless its value has no effect
 * on the format of the data that is written.
 */
static
OkToChangeTag(tif, tag)
    TIFF *tif;
    int tag;
{
    if (tag != TIFFTAG_IMAGELENGTH &&
        (tif->tif_flags & TIFF_BEENWRITING)) {
        TIFFFieldInfo const *fip = TIFFFindFieldInfo(tag, TIFF_ANY);
        /*
         * Consult info table to see if tag can be changed
         * after we've started writing.  We allow changes only
         * to those tags that don't/shouldn't affect the
         * compression and/or format of the data.
         */
        if (fip && !fip->field_oktochange)
            return (0);
```

```
    }
    return (1);
}

/*
 * Record the value of a field in the
 * internal directory structure.  The
 * field will be written to the file
 * when/if the directory structure is
 * updated.
 */
/*VARARGS2*/
DECLARE2V(TIFFSetField, TIFF*, tif, int, tag)
{
    int status = 0;

    if (OkToChangeTag(tif, tag)) {
        va_list ap;

        VA_START(ap, tag);
        status = TIFFSetField1(tif, tag, ap);
        va_end(ap);
    } else {
        TIFFFieldInfo const *fip = TIFFFindFieldInfo(tag, TIFF_ANY);
        if (fip)
            TIFFError("TIFFSetField",
                "%s: Cannot modify tag \"%s\" while writing",
                tif->tif_name, fip->field_name);
    }
    return (status);
}

/*
 * Like TIFFSetField, but taking a varargs
 * parameter list.  This routine is useful
 * for building higher-level interfaces on
 * top of the library.
 */
TIFFVSetField(tif, tag, ap)
    TIFF *tif;
    int tag;
    va_list ap;
{
    int status = 0;

    if (!OkToChangeTag(tif, tag)) {
        TIFFFieldInfo const *fip = TIFFFindFieldInfo(tag, TIFF_ANY);
        if (fip)
            TIFFError("TIFFVSetField",
```

```
                "%s: Cannot modify tag \"%s\" while writing",
                tif->tif_name, fip->field_name);
        } else
            status = TIFFSetField1(tif, tag, ap);
        return (status);
}
```

For reading back field data, TIFFGetField1( ) is the central routine. Its argument is a pointer to the place to store the retrieved data rather than the actual value, but it is otherwise quite similar to TIFFSetField1( ).

```
static
TIFFGetField1(td, tag, ap)
    TIFFDirectory *td;
    int tag;
    va_list ap;
{

    switch (tag) {
    case TIFFTAG_SUBFILETYPE:
        *va_arg(ap, u_long *) = td->td_subfiletype;
        break;
    case TIFFTAG_IMAGEWIDTH:
        *va_arg(ap, u_long *) = td->td_imagewidth;
        break;
    case TIFFTAG_IMAGELENGTH:
        *va_arg(ap, u_long *) = td->td_imagelength;
        break;
    case TIFFTAG_BITSPERSAMPLE:
        *va_arg(ap, u_short *) = td->td_bitspersample;
        break;
    case TIFFTAG_COMPRESSION:
        *va_arg(ap, u_short *) = td->td_compression;
        break;
    case TIFFTAG_PHOTOMETRIC:
        *va_arg(ap, u_short *) = td->td_photometric;
        break;
    case TIFFTAG_THRESHHOLDING:
        *va_arg(ap, u_short *) = td->td_threshholding;
        break;
    case TIFFTAG_FILLORDER:
        *va_arg(ap, u_short *) = td->td_fillorder;
        break;
    case TIFFTAG_DOCUMENTNAME:
        *va_arg(ap, char **) = td->td_documentname;
        break;
    case TIFFTAG_ARTIST:
        *va_arg(ap, char **) = td->td_artist;
        break;
```

```
        case TIFFTAG_DATETIME:
            *va_arg(ap, char **) = td->td_datetime;
            break;
        case TIFFTAG_HOSTCOMPUTER:
            *va_arg(ap, char **) = td->td_hostcomputer;
            break;
        case TIFFTAG_IMAGEDESCRIPTION:
            *va_arg(ap, char **) = td->td_imagedescription;
            break;
        case TIFFTAG_MAKE:
            *va_arg(ap, char **) = td->td_make;
            break;
        case TIFFTAG_MODEL:
            *va_arg(ap, char **) = td->td_model;
            break;
        case TIFFTAG_SOFTWARE:
            *va_arg(ap, char **) = td->td_software;
            break;
        case TIFFTAG_ORIENTATION:
            *va_arg(ap, u_short *) = td->td_orientation;
            break;
        case TIFFTAG_SAMPLESPERPIXEL:
            *va_arg(ap, u_short *) = td->td_samplesperpixel;
            break;
        case TIFFTAG_ROWSPERSTRIP:
            *va_arg(ap, u_long *) = td->td_rowsperstrip;
            break;
        case TIFFTAG_MINSAMPLEVALUE:
            *va_arg(ap, u_short *) = td->td_minsamplevalue;
            break;
        case TIFFTAG_MAXSAMPLEVALUE:
            *va_arg(ap, u_short *) = td->td_maxsamplevalue;
            break;
        case TIFFTAG_XRESOLUTION:
            *va_arg(ap, float *) = td->td_xresolution;
            break;
        case TIFFTAG_YRESOLUTION:
            *va_arg(ap, float *) = td->td_yresolution;
            break;
        case TIFFTAG_PLANARCONFIG:
            *va_arg(ap, u_short *) = td->td_planarconfig;
            break;
        case TIFFTAG_XPOSITION:
            *va_arg(ap, float *) = td->td_xposition;
            break;
        case TIFFTAG_YPOSITION:
            *va_arg(ap, float *) = td->td_yposition;
            break;
        case TIFFTAG_PAGENAME:
```

```
            *va_arg(ap, char **) = td->td_pagename;
            break;
        case TIFFTAG_GROUP3OPTIONS:
            *va_arg(ap, u_long *) = td->td_group3options;
            break;
        case TIFFTAG_GROUP4OPTIONS:
            *va_arg(ap, u_long *) = td->td_group4options;
            break;
        case TIFFTAG_RESOLUTIONUNIT:
            *va_arg(ap, u_short *) = td->td_resolutionunit;
            break;
        case TIFFTAG_PAGENUMBER:
            *va_arg(ap, u_short *) = td->td_pagenumber[0];
            *va_arg(ap, u_short *) = td->td_pagenumber[1];
            break;
        case TIFFTAG_HALFTONEHINTS:
            *va_arg(ap, u_short *) = td->td_halftonehints[0];
            *va_arg(ap, u_short *) = td->td_halftonehints[1];
            break;
        case TIFFTAG_COLORMAP:
            *va_arg(ap, u_short **) = td->td_colormap[0];
            *va_arg(ap, u_short **) = td->td_colormap[1];
            *va_arg(ap, u_short **) = td->td_colormap[2];
            break;
        case TIFFTAG_PREDICTOR:
            *va_arg(ap, u_short *) = td->td_predictor;
            break;
        case TIFFTAG_STRIPOFFSETS:
        case TIFFTAG_TILEOFFSETS:
            *va_arg(ap, u_long **) = td->td_stripoffset;
            break;
        case TIFFTAG_STRIPBYTECOUNTS:
        case TIFFTAG_TILEBYTECOUNTS:
            *va_arg(ap, u_long **) = td->td_stripbytecount;
            break;
        case TIFFTAG_MATTEING:
            *va_arg(ap, u_short *) = td->td_matteing;
            break;
        case TIFFTAG_EXTRASAMPLES:
            *va_arg(ap, u_short *) = td->td_matteing;
            *va_arg(ap, u_short **) = &td->td_matteing;
            break;
        case TIFFTAG_BADFAXLINES:
            *va_arg(ap, u_long *) = td->td_badfaxlines;
            break;
        case TIFFTAG_CLEANFAXDATA:
            *va_arg(ap, u_short *) = td->td_cleanfaxdata;
            break;
        case TIFFTAG_CONSECUTIVEBADFAXLINES:
```

```
            *va_arg(ap, u_long *) = td->td_badfaxrun;
            break;
        case TIFFTAG_TILEWIDTH:
            *va_arg(ap, u_long *) = td->td_tilewidth;
            break;
        case TIFFTAG_TILELENGTH:
            *va_arg(ap, u_long *) = td->td_tilelength;
            break;
        case TIFFTAG_TILEDEPTH:
            *va_arg(ap, u_long *) = td->td_tiledepth;
            break;
        case TIFFTAG_DATATYPE:
            *va_arg(ap, u_short *) =
                (td->td_sampleformat == SAMPLEFORMAT_VOID ?
                0 : td->td_sampleformat);
            break;
        case TIFFTAG_SAMPLEFORMAT:
            *va_arg(ap, u_short *) = td->td_sampleformat;
            break;
        case TIFFTAG_IMAGEDEPTH:
            *va_arg(ap, u_long *) = td->td_imagedepth;
            break;
        default:
            TIFFError("TIFFGetField1",
                "Internal error, no value returned for tag \"%s\"",
                TIFFFieldWithTag(tag)->field_name);
            break;
        }
        va_end(ap);
}
```

The public routines TIFFGetField( ) and TIFFVGetField( ) check to see if the field is set and, if so, they call TIFFGetField1( ) to do the work.

```
/*
 * Return the value of a field in the
 * internal directory structure.
 */
/*VARARGS2*/
DECLARE2V(TIFFGetField, TIFF*, tif, int, tag)
{
    TIFFFieldInfo const *fip = TIFFFindFieldInfo(tag, TIFF_ANY);

    if (fip) {
        u_short bit = fip->field_bit;
        if (bit != FIELD_IGNORE && TIFFFieldSet(tif, bit)) {
            va_list ap;
            VA_START(ap, tag);
            (void) TIFFGetField1(&tif->tif_dir, tag, ap);
```

```
            va_end(ap);
            return (1);
        }
    } else
        TIFFError("TIFFGetField", "Unknown field, tag 0x%x", tag);
    return (0);
}

/*
 * Like TIFFGetField, but taking a varargs
 * parameter list.  This routine is useful
 * for building higher-level interfaces on
 * top of the library.
 */
TIFFVGetField(tif, tag, ap)
    TIFF *tif;
    int tag;
    va_list ap;
{
    TIFFFieldInfo const *fip = TIFFFindFieldInfo(tag, TIFF_ANY);

    if (fip) {
        u_short bit = fip->field_bit;
        if (bit != FIELD_IGNORE && TIFFFieldSet(tif, bit)) {
            (void) TIFFGetField1(&tif->tif_dir, tag, ap);
            return (1);
        }
    } else
        TIFFError("TIFFGetField", "Unknown field, tag 0x%x", tag);
    return (0);
}
```

For use within the library, _TIFFGetField( ) bypasses the validity checks in the public routines.

```
/*
 * Internal interface to TIFFGetField...
 */
int
/*VARARGS2*/
DECLARE2V(_TIFFGetField, TIFFDirectory*, td, int, tag)
{
    va_list ap;

    VA_START(ap, tag);
    (void) TIFFGetField1(td, tag, ap);
    va_end(ap);
}
```

To initialize a directory, we call TIFFDefaultDirectory( ), which sets all of the fields to default values. To get rid of a directory, we call TIFFFreeDirectory( ), which uses the macro CleanupField to free all of the allocated fields pointed to from the directory.

```
/*
 * Set up a default directory structure.
 */
TIFFDefaultDirectory(tif)
    TIFF *tif;
{
    register TIFFDirectory *td = &tif->tif_dir;

    bzero((char *)td, sizeof (*td));
    td->td_fillorder = FILLORDER_MSB2LSB;
    td->td_bitspersample = 1;
    td->td_threshholding = THRESHHOLD_BILEVEL;
    td->td_orientation = ORIENTATION_TOPLEFT;
    td->td_samplesperpixel = 1;
    td->td_predictor = 1;
    td->td_rowsperstrip = 0xffffffff;
    td->td_tilewidth = 0xffffffff;
    td->td_tilelength = 0xffffffff;
    td->td_tiledepth = 1;
    td->td_resolutionunit = RESUNIT_INCH;
    td->td_sampleformat = SAMPLEFORMAT_VOID;
    td->td_imagedepth = 1;
    (void) TIFFSetField(tif, TIFFTAG_COMPRESSION, COMPRESSION_NONE);
    /*
     * NB: The directory is marked dirty as a result of setting
     * up the default compression scheme.  However, this really
     * isn't correct--we want TIFF_DIRTYDIRECT to be set only
     * if the user does something.  We could just do the setup
     * by hand, but it seems better to use the normal mechanism
     * (i.e., TIFFSetField).
     */
    tif->tif_flags &= ~TIFF_DIRTYDIRECT;
    return (1);
}

#define CleanupField(member) {      \
    if (td->member) {               \
    free((char *)td->member);       \
    td->member = 0;                 \
    }                               \
}

/*
 * Release storage associated with a directory. */
```

```
TIFFFreeDirectory(tif)
    TIFF *tif;
{
    register TIFFDirectory *td = &tif->tif_dir;

    CleanupField(td_colormap[0]);
    CleanupField(td_colormap[1]);
    CleanupField(td_colormap[2]);
    CleanupField(td_documentname);
    CleanupField(td_artist);
    CleanupField(td_datetime);
    CleanupField(td_hostcomputer);
    CleanupField(td_imagedescription);
    CleanupField(td_make);
    CleanupField(td_model);
    CleanupField(td_software);
    CleanupField(td_pagename);
    CleanupField(td_stripoffset);
    CleanupField(td_stripbytecount);
}
#undef CleanupField
```

The file tif_aux.c contains two more auxiliary directory handling functions, TIFFVGetFieldDefaulted( ) and TIFFGetFieldDefaulted( ). They act the same as the regular get-field routines, except that if a field was not explicitly set but the directory was initialized to a default value, it returns the default.

```
/*
 * tif_aux.c
 * TIFF Library.
 *
 * Auxiliary Support Routines.
 */
#include "tiffioP.h"
#include "prototypes.h"

/*
 * Like TIFFGetField, but return any default
 * value if the tag is not present in the directory.
 *
 * NB:  We use the value in the directory rather than
 *  explicit values so that defaults exist only one
 *  place in the library--in TIFFDefaultDirectory.
 */
TIFFVGetFieldDefaulted(tif, tag, ap)
    TIFF *tif;
    int tag;
    va_list ap;
```

```
{
    TIFFDirectory *td = &tif->tif_dir;
    int i;

    if (TIFFVGetField(tif, tag, ap))
        return (1);
    switch (tag) {
    case TIFFTAG_SUBFILETYPE:
        *va_arg(ap, u_short *) = td->td_subfiletype;
        return (1);
    case TIFFTAG_BITSPERSAMPLE:
        *va_arg(ap, u_short *) = td->td_bitspersample;
        return (1);
    case TIFFTAG_THRESHHOLDING:
        *va_arg(ap, u_short *) = td->td_threshholding;
        return (1);
    case TIFFTAG_FILLORDER:
        *va_arg(ap, u_short *) = td->td_fillorder;
        return (1);
    case TIFFTAG_ORIENTATION:
        *va_arg(ap, u_short *) = td->td_orientation;
        return (1);
    case TIFFTAG_SAMPLESPERPIXEL:
        *va_arg(ap, u_short *) = td->td_samplesperpixel;
        return (1);
    case TIFFTAG_ROWSPERSTRIP:
        *va_arg(ap, u_long *) = td->td_rowsperstrip;
        return (1);
    case TIFFTAG_MINSAMPLEVALUE:
        *va_arg(ap, u_short *) = td->td_minsamplevalue;
        return (1);
    case TIFFTAG_MAXSAMPLEVALUE:
        *va_arg(ap, u_short *) = td->td_maxsamplevalue;
        return (1);
    case TIFFTAG_PLANARCONFIG:
        *va_arg(ap, u_short *) = td->td_planarconfig;
        return (1);
    case TIFFTAG_GROUP4OPTIONS:
        *va_arg(ap, u_long *) = td->td_group4options;
        return (1);
    case TIFFTAG_RESOLUTIONUNIT:
        *va_arg(ap, u_short *) = td->td_resolutionunit;
        return (1);
    case TIFFTAG_PREDICTOR:
        *va_arg(ap, u_short *) = td->td_predictor;
        return (1);
    case TIFFTAG_TILEDEPTH:
        *va_arg(ap, u_long *) = td->td_tiledepth;
        return (1);
```

```
        case TIFFTAG_DATATYPE:
            *va_arg(ap, u_short *) = td->td_sampleformat-1;
            return (1);
        case TIFFTAG_IMAGEDEPTH:
            *va_arg(ap, u_short *) = td->td_imagedepth;
            return (1);
        }
        return (0);
}

/*
 * Like TIFFGetField, but return any default
 * value if the tag is not present in the directory.
 */
/*VARARGS2*/
DECLARE2V(TIFFGetFieldDefaulted, TIFF*, tif, int, tag)
{
        int ok;
        va_list ap;

        VA_START(ap, tag);
        ok =  TIFFVGetFieldDefaulted(tif, tag, ap);
        va_end(ap);
        return (ok);
}
```

## 18.3 HANDLING TIFF IFDS

On top of the TIFF directory handling code we now build routines that read and write the IFDs in a TIFF file. The on-disk IFD consists of a count of the number of entries in the IFD, the 12-byte entries in increasing tag order, and a pointer to the next IFD, or 0 if this is the last. (Typical TIFF files have only a single image, hence a single IFD.)

### Reading TIFF IFDs

First we declare all of the routines used internally within the file. To save space, we've left out the nonprototype declarations. The routine CheckMalloc( ) is a wrapper around the standard malloc( ), which complains if an allocation fails.

```
/* tif_dirread.c
 * TIFF Library.
 *
 * Directory Read Support Routines.
 *
 * NB: Beware of the varargs declarations for routines in
 *     this file.  The names and types of variables have been
 *     carefully chosen to make things work with compilers that
```

```
 *      are busted in one way or another (e.g., SGI/MIPS).
 */
#include "tiffioP.h"

#define IGNORE  0       /* tag placeholder used below */

#if HAVE_IEEEFP
#define TIFFCvtIEEEFloatToNative(tif, n, fp)    /* dummy */
#endif

#include "prototypes.h"
#if USE_PROTOTYPES
static  EstimateStripByteCounts(TIFF *, TIFFDirEntry *, u_int);
static  MissingRequired(TIFF *, char *);
static  CheckDirCount(TIFF *, TIFFDirEntry *, u_long);
static  TIFFFetchData(TIFF *, TIFFDirEntry *, char *);
static  TIFFFetchString(TIFF *, TIFFDirEntry *, char *);
static  float TIFFFetchRational(TIFF *, TIFFDirEntry *);
static  TIFFFetchNormalTag(TIFF *, TIFFDirEntry *);
static  TIFFFetchPerSampleShorts(TIFF *, TIFFDirEntry *, long *);
static  TIFFFetchShortArray(TIFF *, TIFFDirEntry *, u_short []);
static  TIFFFetchStripThing(TIFF *, TIFFDirEntry *, long,
                                                    u_long **);
static  TIFFFetchRefBlackWhite(TIFF *, TIFFDirEntry *);
static  TIFFFetchJPEGQTables(TIFF *, TIFFDirEntry *);
static  TIFFFetchJPEGCTables(TIFF *, TIFFDirEntry *, u_char ***);
static  TIFFFetchExtraSamples(TIFF *, TIFFDirEntry *);
static  float TIFFFetchFloat(TIFF *, TIFFDirEntry *);
static  int TIFFFetchFloatArray(TIFF *, TIFFDirEntry *, float *);
extern  int TIFFSetCompressionScheme(TIFF *, int);
extern  int TIFFDefaultDirectory(TIFF*);
extern  int TIFFFreeDirectory(TIFF*);
#else
/* nonprototype declarations */
#endif

/* allocate n bytes for purpose "what", complain if it fails
static char *
CheckMalloc(tif, n, what)
    TIFF *tif;
    int n;
    char *what;
{
    char *cp = malloc(n);
    if (cp == NULL)
        TIFFError(tif->tif_name, "No space %s", what);
    return (cp);
}
```

The primary routine to read in a TIFF directory is TIFFReadDirectory( ). It reads from a file that has already been opened by code that we will see later.

Reading the directory is a two-pass process. First it reads the directory from the file and runs down the entries, validating them all and picking up the entries it needs to set the size of variable size tables. Then it allocates the tables and reads the rest of the information from the IFD.

```
/*
 * Read the next TIFF directory from a file
 * and convert it to the internal format.
 * We read directories sequentially.
 */
TIFFReadDirectory(tif)
    TIFF *tif;
{
    register TIFFDirEntry *dp;
    register int n;
    register TIFFDirectory *td;
    TIFFDirEntry *dir;
    long v;
    TIFFFieldInfo *fip;
    u_short dircount;
    char *cp;
    int diroutoforderwarning = 0;
```

Since there can be multiple IFDs, we prepare to get the next one. Normally, the file will just have been opened and the "next" one is really the first one.

```
    tif->tif_diroff = tif->tif_nextdiroff;
                                    /* get next dir in file */
    if (tif->tif_diroff == 0)       /* no more directories */
        return (0);
    tif->tif_curdir++;
```

We seek to the place in the file where the directory is stored, read the count of entries, swap its bytes if the file uses the opposite order of this machine, allocate a buffer, and read in the IFD entries and the pointer to the next IFD.

```
    if (!SeekOK(tif->tif_fd, tif->tif_diroff)) {
        TIFFError(tif->tif_name,
            "Seek error accessing TIFF directory");
        return (0);
    }
    if (!ReadOK(tif->tif_fd, &dircount, sizeof (short))) {
        TIFFError(tif->tif_name,
            "Cannot read TIFF directory count");
        return (0);
    }
```

```
if (tif->tif_flags & TIFF_SWAB)
    TIFFSwabShort(&dircount);
dir = (TIFFDirEntry *)CheckMalloc(tif,
    dircount * sizeof (TIFFDirEntry),
                                    "to read TIFF directory");
if (dir == NULL)
    return (0);
if (!ReadOK(tif->tif_fd, dir, dircount*sizeof
                                        (TIFFDirEntry))) {
    TIFFError(tif->tif_name, "Cannot read TIFF directory");
    goto bad;
}
/*
 * Read offset to next directory for sequential scans.
 */
if (!ReadOK(tif->tif_fd, &tif->tif_nextdiroff,
                                    sizeof (long)))
    tif->tif_nextdiroff = 0;

if (tif->tif_flags & TIFF_SWAB)
    TIFFSwabLong((u_long *)&tif->tif_nextdiroff);
```

Before we to scan the IFD, we free any previous TIFFDirectory structure and create a new one with default entries.

```
tif->tif_flags &= ~TIFF_BEENWRITING; /* reset before new dir */
/*
 * Set up default value and then make a pass over
 * the fields to check type and tag information
 * and to extract info required to size data
 * structures.  A second pass is made afterward
 * to read in everything not taken in the first pass.
 */
td = &tif->tif_dir;
/* free any old stuff and reinit */
TIFFFreeDirectory(tif);
TIFFDefaultDirectory(tif);
/*
 * Electronic Arts writes grayscale TIFF files
 * without a PlanarConfiguration directory entry.
 * Thus we set up a default value here, even though
 * the TIFF spec says there is no default value.
 */
TIFFSetField(tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);
```

Now we iterate through all of the IFD entries, checking that we recognize the tag, that the type is correct, and that the entry has an appropriate number of values. As a comment notes, the tags are supposed to be in ascending order

but some buggy TIFF writers don't sort them. Rejected tags are replaced by the IGNORE tag so we don't have to worry about them on the next pass.

```
for (fip = tiffFieldInfo, dp = dir, n = dircount; n > 0;
        n--, dp++) {
    if (tif->tif_flags & TIFF_SWAB) {
        TIFFSwabArrayOfShort(&dp->tdir_tag, 2);
        TIFFSwabArrayOfLong(&dp->tdir_count, 2);
    }
    /*
     * Find the field information entry for this tag.
     */
    /*
     * Silicon Beach (at least) writes unordered
     * directory tags (violating the spec).  Handle
     * it here, but be obnoxious (maybe they'll fix it?).
     */
    if (dp->tdir_tag < fip->field_tag) {
        if (!diroutoforderwarning) {
            TIFFWarning(tif->tif_name,
"invalid TIFF directory;
 tags are not sorted in ascending order");
            diroutoforderwarning = 1;
        }
        fip = tiffFieldInfo;    /* O(n^2) */
    }
    while (fip->field_tag && fip->field_tag < dp->tdir_tag)
        fip++;
    if (!fip->field_tag || fip->field_tag != dp->tdir_tag) {
        TIFFWarning(tif->tif_name,
            "unknown field with tag %d (0x%x) ignored",
            dp->tdir_tag,  dp->tdir_tag);
        dp->tdir_tag = IGNORE;
        fip = tiffFieldInfo;    /* restart search */
        continue;
    }
    /*
     * Null out old tags that we ignore.
     */
    if (fip->field_bit == FIELD_IGNORE) {
ignore:
        dp->tdir_tag = IGNORE;
        continue;
    }
    /*
     * Check datatype.
     */
    while (dp->tdir_type != (u_short)fip->field_type) {
        if (fip->field_type == TIFF_ANY)    /* wildcard */
```

```
                break;
            fip++;
            if (!fip->field_tag || fip->field_tag != dp->tdir_tag) {
                TIFFWarning(tif->tif_name,
                    "wrong datatype %d for \"%s\"; tag ignored",
                     dp->tdir_type, fip[-1].field_name);
                goto ignore;
            }
        }

        /*
         * Check count if known in advance.
         */
        if (fip->field_readcount != TIFF_VARIABLE) {
            u_long expected = (fip->field_readcount == TIFF_SPP) ?
                (u_long) td->td_samplesperpixel :
                (u_long) fip->field_readcount;
            if (!CheckDirCount(tif, dp, expected))
                goto ignore;
        }

        switch (dp->tdir_tag) {
        case TIFFTAG_STRIPOFFSETS:
        case TIFFTAG_STRIPBYTECOUNTS:
        case TIFFTAG_TILEOFFSETS:
        case TIFFTAG_TILEBYTECOUNTS:
            TIFFSetFieldBit(tif, fip->field_bit);
            break;
        case TIFFTAG_IMAGEWIDTH:
        case TIFFTAG_IMAGELENGTH:
        case TIFFTAG_IMAGEDEPTH:
        case TIFFTAG_TILELENGTH:
        case TIFFTAG_TILEWIDTH:
        case TIFFTAG_TILEDEPTH:
        case TIFFTAG_PLANARCONFIG:
        case TIFFTAG_SAMPLESPERPIXEL:
        case TIFFTAG_ROWSPERSTRIP:
            if (!TIFFFetchNormalTag(tif, dp))
                goto bad;
            break;
        }
    }
```

Now we allocate space for internal buffers. We check that we've read the length and width (jointly marked by the flag FIELD_IMAGEDIMENSIONS) and the planar configuration tag, which tells whether the image data for each pixel is stored together or whether the color components in true-color image are stored separately. (The latter form, though possible, is quite uncommon.)

```
/*
 * Allocate directory structure and set up defaults.
 */
if (!TIFFFieldSet(tif, FIELD_IMAGEDIMENSIONS)) {
    MissingRequired(tif, "ImageLength");
    goto bad;
}
if (!TIFFFieldSet(tif, FIELD_PLANARCONFIG)) {
    MissingRequired(tif, "PlanarConfiguration");
    goto bad;
}
```

If the image is stored in strips with each strip containing full rows of the image, we figure out how many strips are present and set the tile fields to indicate that each "tile" is in fact a full-width strip. If the image is tiled, we call a separate routine (not shown here) to compute the size of the tiles.

```
/*
 * Set up appropriate structures (by strip or by tile)
 */
if (!TIFFFieldSet(tif, FIELD_TILEDIMENSIONS)) {

    /* stored by strip, compute number of strips */
    td->td_stripsperimage = (td->td_rowsperstrip == 0xffffffff ?
         (td->td_imagelength != 0 ? 1 : 0) :
         howmany(td->td_imagelength, td->td_rowsperstrip));
    td->td_tilewidth = td->td_imagewidth;
    td->td_tilelength = td->td_rowsperstrip;
    td->td_tiledepth = td->td_imagedepth;
    tif->tif_flags &= ~TIFF_ISTILED;
} else {
    /* stored by tiles */
    td->td_stripsperimage = TIFFNumberOfTiles(tif);
    tif->tif_flags |= TIFF_ISTILED;
}
td->td_nstrips = td->td_stripsperimage;
if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
    td->td_nstrips *= td->td_samplesperpixel;
```

The mandatory strip offsets field gives the position in the file of each of the strips in the file.

```
if (td->td_nstrips > 0
        && !TIFFFieldSet(tif, FIELD_STRIPOFFSETS)) {
    MissingRequired(tif,
        isTiled(tif) ? "TileOffsets" : "StripOffsets");
    goto bad;
}
```

In the second pass over the IFD, we collect the rest of the data that describes the image, such as the compression technique, the number of bits per pixel, the minimum and maximum values for each pixel (or pixel component), and the actual values of the strip or tile offsets.

```
/*
 * Second pass: extract other information.
 */
for (dp = dir, n = dircount; n > 0; n--, dp++) {
    if (dp->tdir_tag == IGNORE)
        continue;
    switch (dp->tdir_tag) {
    case TIFFTAG_COMPRESSION:
    case TIFFTAG_MINSAMPLEVALUE:
    case TIFFTAG_MAXSAMPLEVALUE:
    case TIFFTAG_BITSPERSAMPLE:
        /*
         * The 5.0 spec says the Compression tag has
         * one value, while earlier specs say it has
         * one value per sample.  Because of this, we
         * accept the tag if one value is supplied.
         *
         * The MinSampleValue, MaxSampleValue, and
         * BitsPerSample tags are supposed to be written
         * as one value/sample, but some vendors incorrectly
         * write one value only--so we accept that
         * as well (yech).
         */
        if (dp->tdir_count == 1) {
            v = TIFFExtractData(tif,
                dp->tdir_type, dp->tdir_offset);
            if (!TIFFSetField(tif, dp->tdir_tag, (int)v))
                goto bad;
            break;
        }
        /* fall thru... */
    case TIFFTAG_DATATYPE:
    case TIFFTAG_SAMPLEFORMAT:
        if (!TIFFFetchPerSampleShorts(tif, dp, &v) ||
            !TIFFSetField(tif, dp->tdir_tag, (int)v))
            goto bad;
        break;
    case TIFFTAG_STRIPOFFSETS:
    case TIFFTAG_TILEOFFSETS:
        if (!TIFFFetchStripThing(tif, dp,
            td->td_nstrips, &td->td_stripoffset))
            goto bad;
        break;
    case TIFFTAG_STRIPBYTECOUNTS:
```

```
case TIFFTAG_TILEBYTECOUNTS:
    if (!TIFFFetchStripThing(tif, dp,
        td->td_nstrips, &td->td_stripbytecount))
        goto bad;
    break;
case TIFFTAG_IMAGELENGTH:
case TIFFTAG_ROWSPERSTRIP:
case TIFFTAG_TILELENGTH:
case TIFFTAG_TILEWIDTH:
case TIFFTAG_TILEDEPTH:
case TIFFTAG_SAMPLESPERPIXEL:
case TIFFTAG_PLANARCONFIG:
    /* handled in first pass above */
    break;
case TIFFTAG_COLORMAP:
    if (!CheckDirCount(tif,dp,3*(1L<<td->td_bitspersample)))
        break;
    /* fall thru... */
case TIFFTAG_TRANSFERFUNCTION:
                                /* gamma or other correction */
    v = (1L<<td->td_bitspersample) * sizeof (u_short);
    cp = CheckMalloc(tif,
        dp->tdir_count * sizeof (u_short),
        "to read \"TransferFunction\" tag");
    if (cp != NULL) {
        if (TIFFFetchData(tif, dp, cp)) {
            /*
             * This deals with the fact that there is only
             * one array to apply to all samples.
             */
            if (dp->tdir_count == 1L<<td->td_bitspersample)
                v = 0;
            /* NB: we assume samples/pixel <= 4 */
            TIFFSetField(tif, dp->tdir_tag,
                cp, cp+v, cp+2*v, cp+3*v);
        }
        free(cp);
    }
    break;
case TIFFTAG_PAGENUMBER:
    if (TIFFFetchShortArray(tif, dp, td->td_pagenumber))
        TIFFSetFieldBit(tif, FIELD_PAGENUMBER);
    break;
case TIFFTAG_HALFTONEHINTS:
    if (TIFFFetchShortArray(tif, dp, td->td_halftonehints))
        TIFFSetFieldBit(tif, FIELD_HALFTONEHINTS);
    break;
case TIFFTAG_EXTRASAMPLES:
    (void) TIFFFetchExtraSamples(tif, dp);
```

```
                break;
/* BEGIN REV 4.0 COMPATIBILITY */
            case TIFFTAG_OSUBFILETYPE:
                v = 0;
                switch (TIFFExtractData(tif, dp->tdir_type,
                    dp->tdir_offset)) {
                case OFILETYPE_REDUCEDIMAGE:
                    v = FILETYPE_REDUCEDIMAGE;
                    break;
                case OFILETYPE_PAGE:
                    v = FILETYPE_PAGE;
                    break;
                }
                if (v)
                    (void) TIFFSetField(tif,
                        TIFFTAG_SUBFILETYPE, (int)v);
                break;
/* END REV 4.0 COMPATIBILITY */
            default:
                (void) TIFFFetchNormalTag(tif, dp);
                break;
            }
        }
```

Now we make some consistency checks. If the image is color-mapped, we check that there is, in fact, a color map. The IFD is supposed to include the size of each strip; if it's missing we estimate it based on the size of the image, if the image data is uncompressed, or otherwise on the size of the file minus the size of the IFD and other descriptor information.

```
    /*
     * Verify Palette image has a Colormap.
     */
    if (td->td_photometric == PHOTOMETRIC_PALETTE &&
        !TIFFFieldSet(tif, FIELD_COLORMAP)) {
        MissingRequired(tif, "Colormap");
        goto bad;
    }
    /*
     * Attempt to deal with a missing StripByteCounts tag.
     */
    if (!TIFFFieldSet(tif, FIELD_STRIPBYTECOUNTS)) {
        /*
         * Some manufacturers violate the spec by not giving
         * the size of the strips.  In this case, assume there
         * is one uncompressed strip of data.
         */
        if (td->td_nstrips > 1) {
            MissingRequired(tif, "StripByteCounts");
```

```
                goto bad;
            }
            TIFFWarning(tif->tif_name,
"TIFF directory is missing required \"%s\" field, calculating from
                                                    imagelength",
                TIFFFieldWithTag(TIFFTAG_STRIPBYTECOUNTS)->field_name);
            EstimateStripByteCounts(tif, dir, dircount);
        } else if (td->td_nstrips == 1
                                && td->td_stripbytecount[0] == 0) {
            /*
             * Plexus (and others) sometimes give a value
             * of 0 for a tag when they don't know what
             * the correct value is!  Try to handle the
             * simple case of estimating the size of a
             * 1-strip image.
             */
            TIFFWarning(tif->tif_name,
"Bogus \"%s\" field, ignoring and calculating from
                                                imagelength",
                TIFFFieldWithTag(TIFFTAG_STRIPBYTECOUNTS)->
                                                field_name);
            EstimateStripByteCounts(tif, dir, dircount);
        }
```

Finally we set a few more defaults: the maximum pixel value defaults to the maximum value that can be represented in the number of bits used, and the compression scheme defaults to none.

```
        if (dir)
            free((char *)dir);
        if (!TIFFFieldSet(tif, FIELD_MAXSAMPLEVALUE))
            td->td_maxsamplevalue = (1L<<td->td_bitspersample)-1;
        /*
         * Set up default compression scheme.
         */
        if (!TIFFFieldSet(tif, FIELD_COMPRESSION))
            TIFFSetField(tif, TIFFTAG_COMPRESSION, COMPRESSION_NONE);
        /*
         * Reinitialize i/o since we are starting on a new directory.
         */
        tif->tif_row = -1;
        tif->tif_curstrip = -1;
        tif->tif_col = -1;
        tif->tif_curtile = -1;
        tif->tif_tilesize = TIFFTileSize(tif);
        tif->tif_scanlinesize = TIFFScanlineSize(tif);
        return (1);
bad:
        if (dir)
```

```
        free((char *)dir);
    return (0);
}
```

If the IFD is missing the byte counts, EstimateStripByteCounts( ) guesses the sizes based on the size of the image or the size of the file.

```
static
EstimateStripByteCounts(tif, dir, dircount)
    TIFF *tif;
    TIFFDirEntry *dir;
    u_int dircount;
{
    register TIFFDirEntry *dp;
    register TIFFDirectory *td = &tif->tif_dir;
    register int n;

    td->td_stripbytecount = (u_long *)
        CheckMalloc(tif, sizeof (u_long),
                                    "for \"StripByteCounts\" array");
    if (td->td_compression != COMPRESSION_NONE) {
        u_long space = sizeof (TIFFHeader)
            + sizeof (short)
            + (dircount * sizeof (TIFFDirEntry))
            + sizeof (long);
        long filesize = TIFFGetFileSize(tif->tif_fd);
        /* calculate amount of space used by indirect values */
        for (dp = dir, n = dircount; n > 0; n--, dp++) {
            int cc = dp->tdir_count * tiffDataWidth[dp->tdir_type];
            if (cc > sizeof (long))
                space += cc;
        }
        td->td_stripbytecount[0] = filesize - space;
        /*
         * This gross hack handles the case where the offset to
         * the strip is past the place where we think the strip
         * should begin.  Since a strip of data must be contiguous,
         * it's safe to assume that we've overestimated the amount
         * of data in the strip and we trim this number back
         * accordingly.
         */
        if (td->td_stripoffset[0] + td->td_stripbytecount[0] >
            filesize)
            td->td_stripbytecount[0] =
                filesize - td->td_stripoffset[0];
    } else {
        u_long rowbytes = howmany(td->td_bitspersample *
            td->td_samplesperpixel * td->td_imagewidth, 8);
        td->td_stripbytecount[0] = td->td_imagelength * rowbytes;
```

```
    }
    TIFFSetFieldBit(tif, FIELD_STRIPBYTECOUNTS);
    if (!TIFFFieldSet(tif, FIELD_ROWSPERSTRIP))
        td->td_rowsperstrip = td->td_imagelength;
}
```

The TIFF reader ends with a few more auxiliary routines. MissingRequired( ) issues a complaint that a required field isn't present. CheckDirCount( ) checks that an IFD field contains the right number of entries.

```
static
MissingRequired(tif, tagname)
    TIFF *tif;
    char *tagname;
{
    TIFFError(tif->tif_name,
        "TIFF directory is missing required \"%s\" field", tagname);
}

/*
 * Check the count field of a directory
 * entry against a known value.  The caller
 * is expected to skip/ignore the tag if
 * there is a mismatch.
 */
static
CheckDirCount(tif, dir, count)
    TIFF *tif;
    TIFFDirEntry *dir;
    u_long count;
{
    if (count != dir->tdir_count) {
        TIFFWarning(tif->tif_name,
"incorrect count for field \"%s\" (%lu, expecting %lu);
                                                    tag ignored",
            TIFFFieldWithTag(dir->tdir_tag)->field_name,
            dir->tdir_count, count);
        return (0);
    }
    return (1);
}
```

The following routines fetch the values pointed to by the IFD from the file. If the file uses the opposite byte order from the local machine, multibyte values have to be byte-swapped. As a special case, values that fit in 4 bytes or less are stored in the IFD itself. RATIONAL fields are stored as the ratio of two integers but are internally converted to floating point.

```
/*
 * Fetch a contiguous directory item.
 */
static
TIFFFetchData(tif, dir, cp)
    TIFF *tif;
    TIFFDirEntry *dir;
    char *cp;
{
    int cc, w;

    w = tiffDataWidth[dir->tdir_type];
    cc = dir->tdir_count * w;
        if (!SeekOK(tif->tif_fd, dir->tdir_offset))
            goto bad;
        if (!ReadOK(tif->tif_fd, cp, cc))
            goto bad;

    if (tif->tif_flags & TIFF_SWAB) {
        switch (dir->tdir_type) {
        case TIFF_SHORT:
        case TIFF_SSHORT:
            TIFFSwabArrayOfShort((u_short *)cp, dir->tdir_count);
            break;
        case TIFF_LONG:
        case TIFF_SLONG:
        case TIFF_FLOAT:
            TIFFSwabArrayOfLong((u_long *)cp, dir->tdir_count);
            break;
        case TIFF_RATIONAL:
        case TIFF_SRATIONAL:
            TIFFSwabArrayOfLong((u_long *)cp, 2*dir->tdir_count);
            break;
        }
    }
    return (cc);
bad:
    TIFFError(tif->tif_name,
        "Error fetching data for field \"%s\"",
        TIFFFieldWithTag(dir->tdir_tag)->field_name);
    return (0);
}
/*
 * Fetch an ASCII item from the file.
 */
static
TIFFFetchString(tif, dir, cp)
    TIFF *tif;
    TIFFDirEntry *dir;
```

```
    char *cp;
{
    if (dir->tdir_count <= 4) {
        u_long l = dir->tdir_offset;
        if (tif->tif_flags & TIFF_SWAB)
            TIFFSwabLong(&l);
        bcopy(&l, cp, dir->tdir_count);
        return (1);
    }
    return (TIFFFetchData(tif, dir, cp));
}

/*
 * Convert numerator+denominator to float.
 */
static int
cvtRational(tif, dir, num, denom, rv)
    TIFF *tif;
    TIFFDirEntry *dir;
    u_long num, denom;
    float *rv;
{
    if (denom == 0) {
        TIFFError(tif->tif_name,
            "%s: Rational with zero denominator (num = %lu)",
            TIFFFieldWithTag(dir->tdir_tag)->field_name, num);
        return (0);
    } else {
        if (dir->tdir_type == TIFF_RATIONAL)
            *rv = ((float)num / (float)denom);
        else
            *rv = ((float)(long)num / (float)(long)denom);
        return (1);
    }
}

/*
 * Fetch a rational item from the file
 * at offset off and return the value
 * as a floating point number.
 */
static float
TIFFFetchRational(tif, dir)
    TIFF *tif;
    TIFFDirEntry *dir;
{
    u_long l[2];
    float v;
```

```
        return (!TIFFFetchData(tif, dir, (char *)l) ||
            !cvtRational(tif, dir, l[0], l[1], &v) ? 1. : v);
}

/*
 * Fetch a single floating point value
 * from the offset field and return it
 * as a native float.
 */
static float
TIFFFetchFloat(tif, dir)
    TIFF *tif;
    TIFFDirEntry *dir;
{
    float v = (float)
        TIFFExtractData(tif, dir->tdir_type, dir->tdir_offset);
    TIFFCvtIEEEFloatToNative(tif, 1, &v);
                                        /* dummy on IEEE systems */
    return (v);
}

/*
 * Fetch an array of BYTE or SBYTE values.
 */
static
TIFFFetchByteArray(tif, dir, v)
    TIFF *tif;
    TIFFDirEntry *dir;
    u_short v[];
{

    if (dir->tdir_count <= 4) {
        /*
         * Extract data from offset field.
         */
        if (tif->tif_header.tiff_magic == TIFF_BIGENDIAN) {
            switch (dir->tdir_count) {
            case 4: v[3] = dir->tdir_offset & 0xff;
            case 3: v[2] = (dir->tdir_offset >> 8) & 0xff;
            case 2: v[1] = (dir->tdir_offset >> 16) & 0xff;
            case 1: v[0] = dir->tdir_offset >> 24;
            }
        } else {
            switch (dir->tdir_count) {
            case 4: v[3] = dir->tdir_offset >> 24;
            case 3: v[2] = (dir->tdir_offset >> 16) & 0xff;
            case 2: v[1] = (dir->tdir_offset >> 8) & 0xff;
            case 1: v[0] = dir->tdir_offset & 0xff;
            }
```

```
            }
            return (1);
        } else
            return (TIFFFetchData(tif, dir, (char *)v));    /* XXX */
}

/*
 * Fetch an array of SHORT or SSHORT values.
 */

static
TIFFFetchShortArray(tif, dir, v)
    TIFF *tif;
    TIFFDirEntry *dir;
    u_short v[];
{
    if (dir->tdir_count <= 2) {
        if (tif->tif_header.tiff_magic == TIFF_BIGENDIAN) {
            switch (dir->tdir_count) {
            case 2: v[1] = dir->tdir_offset & 0xffff;
            case 1: v[0] = dir->tdir_offset >> 16;
            }
        } else {
            switch (dir->tdir_count) {
            case 2: v[1] = dir->tdir_offset >> 16;
            case 1: v[0] = dir->tdir_offset & 0xffff;
            }
        }
        return (1);
    } else
        return (TIFFFetchData(tif, dir, (char *)v));
}

/*
 * Fetch an array of LONG or SLONG values.
 */

static
TIFFFetchLongArray(tif, dir, v)
    TIFF *tif;
    TIFFDirEntry *dir;
    u_long v[];
{
    if (dir->tdir_count == 1) {
        v[0] = dir->tdir_offset;
        return (1);
    } else
        return (TIFFFetchData(tif, dir, (char *)v));
}
```

```
/*
 * Fetch an array of RATIONAL or SRATIONAL values.
 */

static
TIFFFetchRationalArray(tif, dir, v)
    TIFF *tif;
    TIFFDirEntry *dir;
    float v[];
{
    int ok = 0;
    u_long *l;

    l = (u_long *)CheckMalloc(tif,
        dir->tdir_count*tiffDataWidth[dir->tdir_type],
        "to fetch array of rationals");
    if (l) {
        if (TIFFFetchData(tif, dir, (char *)l)) {
            u_long i;
            for (i = 0; i < dir->tdir_count; i++) {
                ok = cvtRational(tif, dir,
                    l[2*i+0], l[2*i+1], &v[i]);
                if (!ok)
                    break;
            }
        }
        free((char *)l);
    }
    return (ok);
}

/*
 * Fetch an array of FLOAT values.
 */

static
TIFFFetchFloatArray(tif, dir, v)
    TIFF *tif;
    TIFFDirEntry *dir;
    float v[];
{
    if (TIFFFetchData(tif, dir, (char *)v)) {
        TIFFCvtIEEEFloatToNative(tif, dir->tdir_count, v);
                                                /* usually dummy */

        return (1);
    } else
        return (0);
}
```

TIFFFetchNormalTag( ), as its name implies, fetches the value(s) for any IFD field that doesn't require special handling. It switches on the datatype to one of the routines above to handle the particular datatype involved.

Fields with multiple values allocate an array for the values and store its address into the directory. Single-value fields store the value itself into the directory.

```
/*
 * Fetch a tag that is not handled by special case code.
 *
 * NB: DOUBLE and UNDEFINED types are not handled.
 */
static
TIFFFetchNormalTag(tif, dp)
    TIFF *tif;
    TIFFDirEntry *dp;
{
    static char mesg[] = "to fetch tag value";
    int ok = 0;

    if (dp->tdir_count > 1) {       /* array of values */
        char *cp = NULL;

        switch (dp->tdir_type) {
        case TIFF_BYTE:
        case TIFF_SBYTE:
            /* NB: always expand BYTE values to shorts */
            cp = CheckMalloc(tif,
                dp->tdir_count * sizeof (u_short), mesg);
            ok = cp && TIFFFetchByteArray(tif, dp, (u_short *)cp);
            break;
        case TIFF_SHORT:
        case TIFF_SSHORT:
            cp = CheckMalloc(tif,
                dp->tdir_count * sizeof (u_short), mesg);
            ok = cp && TIFFFetchShortArray(tif, dp, (u_short *)cp);
            break;
        case TIFF_LONG:
        case TIFF_SLONG:
            cp = CheckMalloc(tif,
                dp->tdir_count * sizeof (u_long), mesg);
            ok = cp && TIFFFetchLongArray(tif, dp, (u_long *)cp);
            break;
        case TIFF_RATIONAL:
        case TIFF_SRATIONAL:
            cp = CheckMalloc(tif,
                dp->tdir_count * sizeof (float), mesg);
            ok = cp && TIFFFetchRationalArray(tif, dp, (float *)cp);
            break;
        case TIFF_FLOAT:
```

```
            cp = CheckMalloc(tif,
                dp->tdir_count * sizeof (float), mesg);
            ok = cp && TIFFFetchFloatArray(tif, dp, (float *)cp);
            break;
        case TIFF_ASCII:
            /*
             * Some vendors write strings w/o the trailing
             * NULL byte, so always append one just in case.
             */
            cp = CheckMalloc(tif, dp->tdir_count+1, mesg);
            if (ok = (cp && TIFFFetchString(tif, dp, cp)))
                cp[dp->tdir_count] = '\0';  /* XXX */
            break;
        }
        if (ok)
            ok = TIFFSetField(tif, dp->tdir_tag, cp);
        if (cp != NULL)
            free(cp);
    } else if (CheckDirCount(tif, dp, 1)) { /* singleton value */
        char c[2];
        switch (dp->tdir_type) {
        case TIFF_BYTE:
        case TIFF_SBYTE:
        case TIFF_SHORT:
        case TIFF_SSHORT:
            ok = TIFFSetField(tif, dp->tdir_tag, (int)
              TIFFExtractData(tif, dp->tdir_type, dp->tdir_offset));
            break;
        case TIFF_LONG:
        case TIFF_SLONG:
            ok = TIFFSetField(tif, dp->tdir_tag, (u_long)
              TIFFExtractData(tif, dp->tdir_type, dp->tdir_offset));
            break;
        case TIFF_RATIONAL:
        case TIFF_SRATIONAL:
            ok = TIFFSetField(tif, dp->tdir_tag,
                TIFFFetchRational(tif, dp));
            break;
        case TIFF_FLOAT:
            ok = TIFFSetField(tif, dp->tdir_tag,
                TIFFFetchFloat(tif, dp));
            break;
        case TIFF_ASCII:
            if (ok = (TIFFFetchString(tif, dp, c))) {
                c[1] = '\0';        /* XXX paranoid */
                ok = TIFFSetField(tif, dp->tdir_tag, c);
            }
            break;
        }
```

```
    }
    return (ok);
}
```

In TIFF files with more than one component per pixel (RGB files, primarily), it is in theory possible to have separate compression schemes for each component. We don't support that, so we check that all the values are the same for per-component fields.

```
/*
 * Fetch samples/pixel short values for
 * the specified tag and verify that
 * all values are the same.
 */
static
TIFFFetchPerSampleShorts(tif, dir, pl)
    TIFF *tif;
    TIFFDirEntry *dir;
    long *pl;
{
    u_short v[4];
    int samples = tif->tif_dir.td_samplesperpixel;

    if (CheckDirCount(tif, dir, (u_long)samples) &&
        TIFFFetchShortArray(tif, dir, v)) {
        int i;
        for (i = 1; i < samples; i++)
            if (v[i] != v[0]) {
                TIFFError(tif->tif_name,
    "Cannot handle different per-sample values for
                                                field \"%s\"",
                    TIFFFieldWithTag(dir->tdir_tag)->field_name);
                return (0);
            }
        *pl = v[0];
        return (1);
    }
    return (0);
}
```

The peculiarly named TIFFFetchStripThing( ) fetches any of the per-strip or per-tile fields from the IFD. It checks that the field has the correct number of entries, allocates space, reads in the data, and if necessary expands short data in the file to long data in memory.

```
/*
 * Fetch a set of offsets or lengths.
 * While this routine says "strips",
```

```
 * in fact it's also used for tiles.
 */
static
TIFFFetchStripThing(tif, dir, nstrips, lpp)
    TIFF *tif;
    TIFFDirEntry *dir;
    long nstrips;
    u_long **lpp;
{
    register u_long *lp;
    int status;

    if (!CheckDirCount(tif, dir, nstrips))
        return (0);
/*
 * Allocate space for strip information.
 */
if (*lpp == NULL &&
    (*lpp = (u_long *)CheckMalloc(tif,
      nstrips * sizeof (u_long), "for strip array")) == NULL)
    return (0);
lp = *lpp;
if (dir->tdir_type == (int)TIFF_SHORT) {
    /*
     * Handle short->long expansion.
     */
    u_short *dp = (u_short *)CheckMalloc(tif,
        dir->tdir_count* sizeof (u_short),
                                    "to fetch strip tag");
    if (dp == NULL)
        return (0);
    if (status = TIFFFetchShortArray(tif, dir, dp)) {
        register u_short *wp = dp;
        while (nstrips-- > 0)
            *lp++ = *wp++;
        }
        free((char *)dp);
    } else
        status = TIFFFetchLongArray(tif, dir, lp);
    return (status);
}
```

Some images have a nonstandard extra sample per component, e.g., ARGB, which gives an alpha value for each pixel. If the extra-samples tag is present, it must have the value EXTRASAMPLE_ASSOCALPHA or we can't handle it.

```
/*
 * Accept matteing-only ExtraSamples tag.
 */
```

```
static
TIFFFetchExtraSamples(tif, dp)
    TIFF *tif;
    TIFFDirEntry *dp;
{
    int type;

    if (dp->tdir_count != 1) {
        TIFFError(tif->tif_name,
            "Cannot handle more than 1 extra sample/pixel");
        return (0);
    }
    type = TIFFExtractData(tif, dp->tdir_type, dp->tdir_offset);
    if (type != EXTRASAMPLE_ASSOCALPHA) {
        TIFFError(tif->tif_name,
            "Can only handle associated-alpha extra samples");
        return (0);
    }
    return (TIFFSetField(tif, TIFFTAG_MATTEING, 1));
}
```

## Writing TIFF IFDs

Writing an IFD is somewhat simpler than reading one. We don't have to handle as many special or error conditions, since we can assume that the in-memory TIFF directory is complete and consistent.

As usual, we start with declarations of local routines.

```
/* tif_dirwrite.c
 * TIFF Library.
 *
 * Directory Write Support Routines.
 *
 * NB: Beware of the varargs declarations for routines in
 *     this file.  The names and types of variables have been
 *     carefully chosen to make things work with compilers that
 *     are busted in one way or another (e.g., SGI/MIPS).
 */
#include "tiffioP.h"
#include "prototypes.h"

#if HAVE_IEEEFP
#define TIFFCvtNativeToIEEEFloat(tif, n, fp)  /* dummy routine */
#endif

#if USE_PROTOTYPES
static  TIFFWriteNormalTag(TIFF*, TIFFDirEntry*, TIFFFieldInfo*);
static  TIFFSetupShortLong(TIFF *, u_short, TIFFDirEntry *,
                                                    u_long);
```

```
static  TIFFSetupShortPair(TIFF *, u_short, TIFFDirEntry *);
static  TIFFWriteRational(TIFF *,
        TIFFDataType, u_short, TIFFDirEntry *, float);
static  TIFFWritePerSampleShorts(TIFF *, u_short,
                                                TIFFDirEntry *);
static  TIFFWriteShortTable(TIFF *, u_short, TIFFDirEntry *,
                                                int, u_short **);
static  TIFFWriteShortArray(TIFF *,
        TIFFDataType, u_short, TIFFDirEntry *, int, u_short *);
static  TIFFWriteLongArray(TIFF *,
        TIFFDataType, u_short, TIFFDirEntry *, int, u_long *);
static  TIFFWriteRationalArray(TIFF *,
        TIFFDataType, u_short, TIFFDirEntry *, int, float *);
static  TIFFWriteFloatArray(TIFF *,
        TIFFDataType, u_short, TIFFDirEntry *, int, float *);
static  TIFFWriteString(TIFF *, u_short, TIFFDirEntry *, char *);
#else
/* nonprototype declarations here */
#endif

/* macro to write two rationals with two calls to write
 *  rational */
#define WriteRationalPair(type, tag1, v1, tag2, v2) {       \
    if (!TIFFWriteRational(tif, type, tag1, dir, v1))   \
        goto bad;                       \
    if (!TIFFWriteRational(tif, type, tag2, dir+1, v2)) \
        goto bad;                       \
    dir++;                              \
}
```

The main routine to write out the TIFF directory is TIFFWriteDirectory( ).

```
static  long dataoff;
/*
 * Write the contents of the current directory
 * to the specified file.  This routine doesn't
 * handle overwriting a directory with auxiliary
 * storage that's been changed.
 */
TIFFWriteDirectory(tif)
    TIFF *tif;
{
    short dircount, tag;
    int nfields, dirsize;
    char *data;
    TIFFFieldInfo *fip;
    TIFFDirEntry *dir;
    TIFFDirectory *td;
    u_long b, fields[sizeof (td->td_fieldsset) / sizeof (u_long)];
```

If the file is opened read-only, don't write anything.

```
if (tif->tif_mode == O_RDONLY)
    return (1);
```

The various compression methods supply a suite of routines that do the actual compression. Some of the routines are to be called after all of the pixel data has been compressed. We call these routines here to be sure the image is entirely written and the directory is completely updated.

```
/*
 * Clear write state so that subsequent images with
 * different characteristics get the right buffers
 * set up for them.
 */
if (tif->tif_flags & TIFF_POSTENCODE) {
    tif->tif_flags &= ~TIFF_POSTENCODE;
    if (tif->tif_postencode && !(*tif->tif_postencode)(tif)) {
        TIFFError(tif->tif_name,
            "Error post-encoding before directory write");
        return (0);
    }
}
if (tif->tif_close)
    (*tif->tif_close)(tif);
if (tif->tif_cleanup)
    (*tif->tif_cleanup)(tif);
```

Write out any data those routines might have left in internal buffers.

```
/*
 * Flush any data that might have been written
 * by the compression close+cleanup routines.
 */
if (tif->tif_rawcc > 0 && !TIFFFlushData1(tif)) {
    TIFFError(tif->tif_name,
        "Error flushing data before directory write");
    return (0);
}
if ((tif->tif_flags & TIFF_MYBUFFER) && tif->tif_rawdata) {
    free(tif->tif_rawdata);
    tif->tif_rawdata = NULL;
    tif->tif_rawcc = 0;
}
tif->tif_flags &= ~(TIFF_BEENWRITING|TIFF_BUFFERSETUP);
```

The IFD contains pointers to data stored after the IFD. We need to know how big the IFD itself will be so that we can compute the locations of the stored data.

We compute that size here. (If we're updating an existing TIFF file, there is some code to replace the existing directory. In the normal case, the directory goes after what's been written so far.)

We make a temporary copy in fields[ ] of the field-valid bits in the directory, and check a special case about the extra samples tag.

```
td = &tif->tif_dir;
/*
 * Size the directory so that we can calculate
 * offsets for the data items that aren't kept
 * in-place in each field.
 */
nfields = 0;
for (b = 0; b <= FIELD_LAST; b++)
    if (TIFFFieldSet(tif, b))
        nfields += (b < FIELD_SUBFILETYPE ? 2 : 1);
dirsize = nfields * sizeof (TIFFDirEntry);
data = malloc(dirsize);
if (data == NULL) {
    TIFFError(tif->tif_name,
        "Cannot write directory, out of space");
    return (0);
}
/*
 * Directory hasn't been placed yet, put
 * it at the end of the file and link it
 * into the existing directory structure.
 */
if (tif->tif_diroff == 0 && !TIFFLinkDirectory(tif))
    return (0);
dataoff = tif->tif_diroff + sizeof (short) + dirsize +
          sizeof (long);
if (dataoff & 1)
    dataoff++;
(void) lseek(tif->tif_fd, dataoff, L_SET);
tif->tif_curdir++;
dir = (TIFFDirEntry *)data;
/*
 * Set up external form of directory
 * entries and write data items.
 */
 /* fields[] is temporary copy of the field flag bits in the
  * TIFF directory
  */
bcopy(td->td_fieldsset, fields, sizeof (fields));
/*
 * Write out ExtraSamples tag only if Matteing would
 * be set to 1 (i.e., Associated Alpha data is present).
 */
```

```
    if (FieldSet(fields, FIELD_MATTEING) && !td->td_matteing)
        { /*XXX*/
        ResetFieldBit(fields, FIELD_MATTEING);          /*XXX*/
        nfields--;                              /*XXX*/
        dirsize -= sizeof (TIFFDirEntry);           /*XXX*/
    }                                       /*XXX*/
```

Now we iterate through the array tiffFieldInfo, checking each possible field in the IFD. For each field that has valid data we switch to appropriate code to write the field out. All of the actual writing is done by auxiliary routines, which will be shown later.

```
    for (fip = tiffFieldInfo; fip->field_tag; fip++) {
        if (fip->field_bit == FIELD_IGNORE ||
            !FieldSet(fields, fip->field_bit))
            continue;
        switch (fip->field_bit) {
        case FIELD_STRIPOFFSETS:
            /*
             * We use one field bit for both strip and tile
             * offsets, so we must be careful in selecting
             * the appropriate field descriptor (so that tags
             * are written in sorted order).
             */
            tag = isTiled(tif) ?
                TIFFTAG_TILEOFFSETS : TIFFTAG_STRIPOFFSETS;
            if (tag != fip->field_tag)
                continue;
            if (!TIFFWriteLongArray(tif, TIFF_LONG, tag, dir,
                (int) td->td_nstrips, td->td_stripoffset))
                goto bad;
            break;
        case FIELD_STRIPBYTECOUNTS:
            /*
             * We use one field bit for both strip and tile
             * byte counts, so we must be careful in selecting
             * the appropriate field descriptor (so that tags
             * are written in sorted order).
             */
            tag = isTiled(tif) ?
                TIFFTAG_TILEBYTECOUNTS : TIFFTAG_STRIPBYTECOUNTS;
            if (tag != fip->field_tag)
                continue;
            if (!TIFFWriteLongArray(tif, TIFF_LONG, tag, dir,
                (int) td->td_nstrips, td->td_stripbytecount))
                goto bad;
            break;
        case FIELD_COLORMAP:
            if (!TIFFWriteShortTable(tif, TIFFTAG_COLORMAP, dir,
```

```
                3, td->td_colormap))
                goto bad;
            break;
        case FIELD_IMAGEDIMENSIONS:
            TIFFSetupShortLong(tif, TIFFTAG_IMAGEWIDTH,
                dir++, td->td_imagewidth);
            TIFFSetupShortLong(tif, TIFFTAG_IMAGELENGTH,
                dir, td->td_imagelength);
            break;
        case FIELD_TILEDIMENSIONS:
            TIFFSetupShortLong(tif, TIFFTAG_TILEWIDTH,
                dir++, td->td_tilewidth);
            TIFFSetupShortLong(tif, TIFFTAG_TILELENGTH,
                dir, td->td_tilelength);
            break;
        case FIELD_POSITION:
            WriteRationalPair(TIFF_RATIONAL,
                TIFFTAG_XPOSITION, td->td_xposition,
                TIFFTAG_YPOSITION, td->td_yposition);
            break;
        case FIELD_RESOLUTION:
            WriteRationalPair(TIFF_RATIONAL,
                TIFFTAG_XRESOLUTION, td->td_xresolution,
                TIFFTAG_YRESOLUTION, td->td_yresolution);
            break;
        case FIELD_BITSPERSAMPLE:
        case FIELD_MINSAMPLEVALUE:
        case FIELD_MAXSAMPLEVALUE:
        case FIELD_SAMPLEFORMAT:
            if (!TIFFWritePerSampleShorts(tif, fip->field_tag, dir))
                goto bad;
            break;
        case FIELD_PAGENUMBER:
        case FIELD_HALFTONEHINTS:
            TIFFSetupShortPair(tif, fip->field_tag, dir);
            break;
        default:
            if (!TIFFWriteNormalTag(tif, dir, fip))
                goto bad;
            break;
        }
        dir++;
        ResetFieldBit(fields, fip->field_bit);
    }
```

We've written all the fields and updated the directory with the file offsets, so we seek back to the place where the IFD itself goes and write it out.

```
    /*
     * Write directory.
     */
    (void) lseek(tif->tif_fd, tif->tif_diroff, L_SET);
    dircount = nfields;
    if (!WriteOK(tif->tif_fd, &dircount, sizeof (short))) {
        TIFFError(tif->tif_name, "Error writing directory count");
        goto bad;
    }
    if (!WriteOK(tif->tif_fd, data, dirsize)) {
        TIFFError(tif->tif_name,
        "Error writing directory contents");
        goto bad;
    }
    if (!WriteOK(tif->tif_fd,
        &tif->tif_nextdiroff, sizeof (long))) {
        TIFFError(tif->tif_name, "Error writing directory link");
        goto bad;
    }
    TIFFFreeDirectory(tif);
    free(data);
    tif->tif_flags &= ~TIFF_DIRTYDIRECT;
                                /* file agrees with in-mem dir */
```

All done; reset state in case there's another IFD.

```
    /*
     * Reset directory-related state for subsequent
     * directories.
     */
    TIFFDefaultDirectory(tif);
    tif->tif_diroff = 0;
    tif->tif_curoff = 0;
    tif->tif_row = -1;
    tif->tif_curstrip = -1;
    return (1);
bad:
    free(data);
    return (0);
}
```

A somewhat large set of auxiliary routines handles the actual writing of data fields. TIFFWriteNormalTag( ) switches on the size of the data, based on the type. It picks up the nominal field count from TIFFFieldInfo[ ]. If the count can be variable it gets the actual count (typically the length of a string item). Then it calls the appropriate routine, based on the type of the data, to write the data to the file.

```
#undef WriteRationalPair

/*
 * Process tags that are not special-cased.
 */
static
DECLARE3(TIFFWriteNormalTag,
    TIFF*, tif, TIFFDirEntry*, dir, TIFFFieldInfo*, fip)
{
    TIFFDirectory* td = &tif->tif_dir;
    u_short wc = (u_short) fip->field_writecount;

    dir->tdir_tag = fip->field_tag;
    dir->tdir_type = (u_short)fip->field_type;
    dir->tdir_count = wc;
#define WRITE(x,y)  x(tif, fip->field_type, \
    fip->field_tag, dir, wc, y)
    switch (fip->field_type) {
    case TIFF_SHORT:
    case TIFF_SSHORT:
        if (wc > 1) {
            u_short *wp;
            if (wc == (u_short) TIFF_VARIABLE) {
                _TIFFgetfield(td, fip->field_tag, &wc, &wp);
                dir->tdir_count = wc;
            } else
                _TIFFgetfield(td, fip->field_tag, &wp);
            if (!WRITE(TIFFWriteShortArray, wp))
                return (0);
        } else {
            u_short sv;
            _TIFFgetfield(td, fip->field_tag, &sv);
            dir->tdir_offset =
                TIFFInsertData(tif, dir->tdir_type, sv);
        }
        break;
    case TIFF_LONG:
    case TIFF_SLONG:
        if (wc > 1) {
            u_long *lp;
            if (wc == (u_short) TIFF_VARIABLE) {
                _TIFFgetfield(td, fip->field_tag, &wc, &lp);
                dir->tdir_count = wc;
            } else
                _TIFFgetfield(td, fip->field_tag, &lp);
            if (!WRITE(TIFFWriteLongArray, lp))
                return (0);
```

```
        } else {
                /* XXX handle LONG->SHORT conversion */
                _TIFFgetfield(td, fip->field_tag, &dir->tdir_offset);
            }
            break;
        case TIFF_RATIONAL:
        case TIFF_SRATIONAL:
            if (wc > 1) {
                float *fp;
                if (wc == (u_short) TIFF_VARIABLE) {
                    _TIFFgetfield(td, fip->field_tag, &wc, &fp);
                    dir->tdir_count = wc;
                } else
                    _TIFFgetfield(td, fip->field_tag, &fp);
                if (!WRITE(TIFFWriteRationalArray, fp))
                    return (0);
            } else {
                float fv;
                _TIFFgetfield(td, fip->field_tag, &fv);
                if (!TIFFWriteRational(tif, fip->field_type,
                fip->field_tag, dir, fv))
                    return (0);
            }
            break;
        case TIFF_FLOAT:
            if (wc > 1) {
                float *fp;
                if (wc == (u_short) TIFF_VARIABLE) {
                    _TIFFgetfield(td, fip->field_tag, &wc, &fp);
                    dir->tdir_count = wc;
                } else
                    _TIFFgetfield(td, fip->field_tag, &fp);
                if (!WRITE(TIFFWriteFloatArray, fp))
                    return (0);
            } else {
                float fv;
                _TIFFgetfield(td, fip->field_tag, &fv);
                TIFFCvtNativeToIEEEFloat(tif, 1, &fv);
                /* XXX assumes sizeof (long) == sizeof (float) */
                dir->tdir_offset = *(u_long *)&fv;  /* XXX */
            }
            break;
        case TIFF_ASCII: {
            char *cp;
            _TIFFgetfield(td, fip->field_tag, &cp);
            if (!TIFFWriteString(tif, fip->field_tag, dir, cp))
                return (0);
            break;
        }
```

```
    }
    return (1);
}
#undef WRITE
```

Some IFD fields can be either 16 bit shorts or 32 bit longs. This routine checks to see if all of the data fits in 16 bits, in which case we'll store it as short, or if not, we'll store it as long.

```
/*
 * Set up a directory entry with either a SHORT
 * or LONG type according to the value.
 */
static
DECLARE4(TIFFSetupShortLong,
    TIFF*, tif, u_short, tag, TIFFDirEntry*, dir, u_long, v)
{
    dir->tdir_tag = tag;
    dir->tdir_count = 1;
    if (v > 0xffffL) {
        dir->tdir_type = (short)TIFF_LONG;
        dir->tdir_offset = v;
    } else {
        dir->tdir_type = (short)TIFF_SHORT;
        dir->tdir_offset = TIFFInsertData(tif, (int)TIFF_SHORT, v);
    }
}
#undef MakeShortDirent
```

For rationals, we have to convert from floating point back to a ratio of integers. Rather than use an exact algorithm, which is very slow, we always use a denominator of 10,000 and scale the numerator to provide the correct value. Then we write the actual data.

```
/*
 * Set up a RATIONAL directory entry and
 * write the associated indirect value.
 */
static
DECLARE5(TIFFWriteRational,
    TIFF*, tif, TIFFDataType, type, u_short, tag, TIFFDirEntry*,
    dir,
    float, v)
{
    u_long t[2];

    dir->tdir_tag = tag;
    dir->tdir_type = (short)type;
```

```
    dir->tdir_count = 1;
    if (type == TIFF_RATIONAL && v < 0)
        TIFFWarning(tif->tif_name,
    "\"%s\": Information lost writing value (%g) as (unsigned)
            RATIONAL",
           TIFFFieldWithTag(tag)->field_name, v);
    /* need algorithm to convert ... XXX */
    t[0] = v*10000.0 + 0.5;
    t[1] = 10000;
    return (TIFFWriteData(tif, dir, (char *)t));
}
```

For the fields that are nominally kept per-sample, i.e., per-component, such as bits-per-sample, we replicate the single value we keep so that the correct number of copies is stored in the IFD.

For IFD entries whose values are a pair of shorts (none of great interest), TIFFSetupShortPair( ) picks up the values and writes them, using the more general TIFFWriteShortArray( ), shown later.

```
/*
 * Set up a directory entry that references a samples/pixel array
 * of SHORT values and (potentially) writes the associated indirect
 * values.
 */
static
DECLARE3(TIFFWritePerSampleShorts,
    TIFF*, tif, u_short, tag, TIFFDirEntry*, dir)
{
    u_short w[4], v;
    int i, samplesperpixel = tif->tif_dir.td_samplesperpixel;

    _TIFFgetfield(&tif->tif_dir, tag, &v);
    for (i = 0; i < samplesperpixel; i++)
        w[i] = v;
    return (TIFFWriteShortArray(
        tif, TIFF_SHORT, tag, dir, samplesperpixel, w));
}

/*
 * Set up a pair of shorts that are returned by value rather
 * than as a reference to an array.
 */
static
DECLARE3(TIFFSetupShortPair,
    TIFF*, tif, u_short, tag, TIFFDirEntry*, dir)
{
    u_short v[2];

    _TIFFgetfield(&tif->tif_dir, tag, &v[0], &v[1]);
    return (TIFFWriteShortArray(tif, TIFF_SHORT, tag, dir, 2, v));
}
```

The color map has three short entries per possible pixel value (one each of red, green, and blue). Here we write out the color map as three arrays of shorts.

```
/*
 * Set up a directory entry for an NxM table of shorts,
 * where M is known to be 2**bitspersample, and write
 * the associated indirect data.
 */
static
DECLARE5(TIFFWriteShortTable,
    TIFF*, tif, u_short, tag, TIFFDirEntry*, dir, int, n,
    u_short**, table)
{
    u_long off;
    int i;

    dir->tdir_tag = tag;
    dir->tdir_type = (short)TIFF_SHORT;
    /* XXX -- adjust for TIFFWriteData's benefit */
    dir->tdir_count = 1L<<tif->tif_dir.td_bitspersample;
    off = dataoff;
    for (i = 0; i < n; i++)
        if (!TIFFWriteData(tif, dir, (char *)table[i]))
            return (0);
    dir->tdir_count *= n;
    dir->tdir_offset = off;
    return (1);
}
```

For ASCII strings, if the string is four characters or less, it fits in the IFD, so we needn't write anything here. If it's longer, we write it out.

```
/*
 * Set up a directory entry of an ASCII string
 * and write any associated indirect value.
 */
static
DECLARE4(TIFFWriteString,
    TIFF*, tif, u_short, tag, TIFFDirEntry*, dir, char*, cp)
{
    dir->tdir_tag = tag;
    dir->tdir_type = (short)TIFF_ASCII;
    dir->tdir_count = strlen(cp) + 1;    /* includes \0 byte */
    if (dir->tdir_count > 4) {
        if (!TIFFWriteData(tif, dir, cp))
            return (0);
    } else
        bcopy(cp, &dir->tdir_offset, dir->tdir_count);
    return (1);
}
```

The same principle applies for arrays of longs or shorts: two or less shorts or a single long fit in the IFD, but longer arrays are written now.

```
/*
 * Set up a directory entry of an array of SHORT
 * or SSHORT and write the associated indirect values.
 */
static
DECLARE6(TIFFWriteShortArray, TIFF*, tif,
    TIFFDataType, type, u_short, tag, TIFFDirEntry*, dir, int,
    n, u_short*, v)
{
    dir->tdir_tag = tag;
    dir->tdir_type = (short)type;
    dir->tdir_count = n;
    if (n <= 2) {
        if (tif->tif_header.tiff_magic == TIFF_BIGENDIAN) {
            dir->tdir_offset = (long)v[0] << 16;
            if (n == 2)
                dir->tdir_offset |= v[1] & 0xffff;
        } else {
            dir->tdir_offset = v[0] & 0xffff;
            if (n == 2)
                dir->tdir_offset |= (long)v[1] << 16;
        }
        return (1);
    } else
        return (TIFFWriteData(tif, dir, (char *)v));
}

/*
 * Set up a directory entry of an array of LONG
 * or SLONG and write the associated indirect values.
 */
static
DECLARE6(TIFFWriteLongArray, TIFF*, tif,
    TIFFDataType, type, u_short, tag, TIFFDirEntry*, dir,
    int, n, u_long*, v)
{
    dir->tdir_tag = tag;
    dir->tdir_type = (short)type;
    dir->tdir_count = n;
    if (n == 1) {
        dir->tdir_offset = v[0];
        return (1);
    } else
        return (TIFFWriteData(tif, dir, (char *)v));
}
```

Arrays of rational values, internally stored as floating point, are turned into arrays of pairs of longs, as with the single rationals we saw above, and are written here.

Arrays of floats are written just like arrays of longs, after conversion from internal to IEEE floating point.

```
/*
 * Set up a directory entry of an array of RATIONAL
 * or SRATIONAL and write the associated indirect values.
 */
static
DECLARE6(TIFFWriteRationalArray, TIFF*, tif,
    TIFFDataType, type, u_short, tag, TIFFDirEntry*, dir, int,
    n, float*, v)
{
    int i, status;
    u_long *t;

    dir->tdir_tag = tag;
    dir->tdir_type = (short)type;
    dir->tdir_count = n;
    t = (u_long *)malloc(2*n * sizeof (long));
    for (i = 0; i < n; i++) {
        /* need algorithm to convert ... XXX */
        t[2*i+0] = v[i]*10000.0 + 0.5;
        t[2*i+1] = 10000;
    }
    status = TIFFWriteData(tif, dir, (char *)t);
    free((char *)t);
    return (status);
}

static
DECLARE6(TIFFWriteFloatArray, TIFF *, tif,
    TIFFDataType, type, u_short, tag, TIFFDirEntry *, dir, int,
    n, float *, v)
{
    dir->tdir_tag = tag;
    dir->tdir_type = (short)type;
    dir->tdir_count = n;
    TIFFCvtNativeToIEEEFloat(tif, n, v);
                                    /* dummy on IEEE machines */
    if (n == 1) {
        dir->tdir_offset = *(u_long *)&v[0];
        return (1);
    } else
        return (TIFFWriteData(tif, dir, (char *)v));
}
```

TIFFWriteData( ) actually writes an item to the file. It notes dataoff, the place where the item will be written in the in-memory directory, seeks to the right place, and writes the data. If the length is odd, it rounds the new offset to the next even value, per the TIFF spec.

```
/*
 * Write a contiguous directory item.
 */

static
TIFFWriteData(tif, dir, cp)
    TIFF *tif;
    TIFFDirEntry *dir;
    char *cp;
{
    int cc;

    dir->tdir_offset = dataoff;
    cc = dir->tdir_count * tiffDataWidth[dir->tdir_type];
    if (SeekOK(tif->tif_fd, dir->tdir_offset) &&
        WriteOK(tif->tif_fd, cp, cc)) {
        dataoff += (cc + 1) & ~1;
        return (1);
    }
    TIFFError(tif->tif_name, "Error writing data for field \"%s\"",
        TIFFFieldWithTag(dir->tdir_tag)->field_name);
    return (0);
}
```

The final routine, TIFFLinkDirectory( ), links the new IFD into the chain in the file. If this is the first IFD, the most common case, it updates the TIFF file header to point to this IFD. If not, it updates the pointer at the end of the previous IFD.

```
/*
 * Link the current directory into the
 * directory chain for the file.
 */

static
TIFFLinkDirectory(tif)
    register TIFF *tif;
{
    static char module[] = "TIFFLinkDirectory";
    u_short dircount;
    long nextdir;

    tif->tif_diroff = (lseek(tif->tif_fd, 0L, L_XTND)+1) &~ 1L;
    if (tif->tif_header.tiff_diroff == 0) {
```

```
        /* First directory, overwrite header.
         */
        tif->tif_header.tiff_diroff = tif->tif_diroff;
        (void) lseek(tif->tif_fd, 0L, L_SET);
        if (!WriteOK(tif->tif_fd, &tif->tif_header,
            sizeof (tif->tif_header))) {
            TIFFError(tif->tif_name, "Error writing TIFF header");
            return (0);
        }
        return (1);
    }
    /*
     * Not the first directory, search to the last and append.
     */
    nextdir = tif->tif_header.tiff_diroff;
    do {
        if (!SeekOK(tif->tif_fd, nextdir) ||
            !ReadOK(tif->tif_fd, &dircount, sizeof (dircount))) {
            TIFFError(module, "Error fetching directory count");
            return (0);
        }
        if (tif->tif_flags & TIFF_SWAB)
            TIFFSwabShort(&dircount);
        lseek(tif->tif_fd, dircount * sizeof (TIFFDirEntry),
        L_INCR);
        if (!ReadOK(tif->tif_fd, &nextdir, sizeof (nextdir))) {
            TIFFError(module, "Error fetching directory link");
            return (0);
        }
        if (tif->tif_flags & TIFF_SWAB)
            TIFFSwabLong((u_long *)&nextdir);
    } while (nextdir != 0);
    (void) lseek(tif->tif_fd, -sizeof (nextdir), L_INCR);
    if (!WriteOK(tif->tif_fd,
        &tif->tif_diroff, sizeof (tif->tif_diroff))) {
            TIFFError(module, "Error writing directory link");
            return (0);
    }
    return (1);
}
```

## 18.4 OPENING AND CLOSING TIFF FILES

We go a conceptual step up and look now at the code that opens and closes TIFF files. For the most part, opening a TIFF file consists of initializing the structures handled by the routines we've seen earlier.

The auxiliary routine TIFFInitOrder( ) sets up the type shift and mask arrays used by TIFFExtractData( ) and TIFFInsertData( ) as well as the TIFF_SWAB

flag that tells whether multibyte items need to be byte-swapped (if the file byte order disagrees with the native one).

Following that is getMode( ), which decodes an open mode string similar to that passed to fopen( ) into the bits to pass to the open( ) system call.

```
/* tif_open.c
 * TIFF Library.
 */
#include "tiffioP.h"
#include "prototypes.h"

#if USE_PROTOTYPES
extern  int TIFFDefaultDirectory(TIFF*);
#else
extern  int TIFFDefaultDirectory(~);
#endif

static const long typemask[13] = {
    0,      /* TIFF_NOTYPE */
    0x000000ff, /* TIFF_BYTE */
    0xffffffff, /* TIFF_ASCII */
    0x0000ffff, /* TIFF_SHORT */
    0xffffffff, /* TIFF_LONG */
    0xffffffff, /* TIFF_RATIONAL */
    0x000000ff, /* TIFF_SBYTE */
    0x000000ff, /* TIFF_UNDEFINED */
    0x0000ffff, /* TIFF_SSHORT */
    0xffffffff, /* TIFF_SLONG */
    0xffffffff, /* TIFF_SRATIONAL */
    0xffffffff, /* TIFF_FLOAT */
    0xffffffff, /* TIFF_DOUBLE */
};
static const int bigTypeshift[13] = {
    0,      /* TIFF_NOTYPE */
    24,     /* TIFF_BYTE */
    0,      /* TIFF_ASCII */
    16,     /* TIFF_SHORT */
    0,      /* TIFF_LONG */
    0,      /* TIFF_RATIONAL */
    16,     /* TIFF_SBYTE */
    16,     /* TIFF_UNDEFINED */
    24,     /* TIFF_SSHORT */
    0,      /* TIFF_SLONG */
    0,      /* TIFF_SRATIONAL */
    0,      /* TIFF_FLOAT */
    0,      /* TIFF_DOUBLE */
};
static const int litTypeshift[13] = {
    0,      /* TIFF_NOTYPE */
```

```
        0,      /* TIFF_BYTE */
        0,      /* TIFF_ASCII */
        0,      /* TIFF_SHORT */
        0,      /* TIFF_LONG */
        0,      /* TIFF_RATIONAL */
        0,      /* TIFF_SBYTE */
        0,      /* TIFF_UNDEFINED */
        0,      /* TIFF_SSHORT */
        0,      /* TIFF_SLONG */
        0,      /* TIFF_SRATIONAL */
        0,      /* TIFF_FLOAT */
        0,      /* TIFF_DOUBLE */
};

/*
 * Initialize the bit fill order, the
 * shift & mask tables, and the byte
 * swapping state according to the file
 * contents and the machine architecture.
 */
static
DECLARE3(TIFFInitOrder, register TIFF*, tif, int, magic,
int, bigendian)
{
    /* XXX how can we deduce this dynamically? */
    tif->tif_fillorder = FILLORDER_MSB2LSB;

    tif->tif_typemask = typemask;
    if (magic == TIFF_BIGENDIAN) {
        tif->tif_typeshift = bigTypeshift;
        if (!bigendian)
            tif->tif_flags |= TIFF_SWAB;
    } else {
        tif->tif_typeshift = litTypeshift;
        if (bigendian)
            tif->tif_flags |= TIFF_SWAB;
    }
}

static int
DECLARE2(getMode, char*, mode, char*, module)
{
    int m = -1;

    switch (mode[0]) {
    case 'r':
        m = O_RDONLY;
        if (mode[1] == '+')
            m = O_RDWR;
```

```
            break;
        case 'w':
        case 'a':
            m = O_RDWR|O_CREAT;
            if (mode[0] == 'w')
                m |= O_TRUNC;
            break;
        default:
            TIFFError(module, "\"%s\": Bad mode", mode);
            break;
        }
        return (m);
}
```

TIFFOpen( ) is the usual entry point for opening a TIFF file to read or write. It decodes the mode string, saves a copy of the file name, opens the file using TIFFOpenFile( ), a macro wrapped around the open( ) system call, and then calls TIFFFdOpen( ) to do the rest of the work.

```
/*
 * Open a TIFF file for read/writing.
 */
TIFF *
TIFFOpen(name, mode)
    char *name, *mode;
{
    static char module[] = "TIFFOpen";
    int m, fd;

    m = getMode(mode, module);
    if (m == -1)
        return ((TIFF *)0);
    fd = TIFFOpenFile(name, m, 0666);
    if (fd < 0) {
        TIFFError(module, "%s: Cannot open", name);
        return ((TIFF *)0);
    }
    return (TIFFFdOpen(fd, name, mode));
}
```

TIFFFdOpen( ) is analogous to the fdopen( ) stdio call. It takes a file that is already open and prepares to use it for TIFF I/O. After checking the mode string, it allocates and initializes a TIFF data structure to handle the file.

```
/*
 * Open a TIFF file descriptor for read/writing.
 */
TIFF *
```

```
TIFFFdOpen(fd, name, mode)
    int fd;
    char *name, *mode;
{
    static char module[] = "TIFFFdOpen";
    TIFF *tif;
    int m, bigendian;

    m = getMode(mode, module);
    if (m == -1)
        goto bad2;
    tif = (TIFF *)malloc(sizeof (TIFF) + strlen(name) + 1);
    if (tif == NULL) {
        TIFFError(module, "%s: Out of memory (TIFF structure)",
        name);
        goto bad2;
    }
    bzero((char *)tif, sizeof (*tif));
    tif->tif_name = (char *)tif + sizeof (TIFF);
    strcpy(tif->tif_name, name);
    tif->tif_fd = fd;
    tif->tif_mode = m &~ (O_CREAT|O_TRUNC);
    tif->tif_curdir = -1;        /* nonexistent directory */
    tif->tif_curoff = 0;
    tif->tif_curstrip = -1;      /* invalid strip */
    tif->tif_row = -1;       /* read/write pre-increment */

    /* check whether this is a big- or little-endian system */

    { int one = 1; bigendian = (*(char *)&one == 0); }
```

Then it tries to read in the TIFF header, which indicates the byte order used in the file and points to the first IFD.

If the file is open for writing and there is no TIFF header, we write one now. In preparation for writing the IFD, a call to TIFFDefaultDirectory creates a TIFF directory structure to use for the file and sets it to default values. That completes the open process for a new file.

```
    /*
     * Read in TIFF header.
     */
    if (!ReadOK(fd, &tif->tif_header, sizeof (TIFFHeader))) {
        if (tif->tif_mode == O_RDONLY) {
            TIFFError(name, "Cannot read TIFF header");
            goto bad;
        }
        /*
         * Set up header and write.
         */
```

```
    /* use local byte order */
    tif->tif_header.tiff_magic =  bigendian ?
        TIFF_BIGENDIAN : TIFF_LITTLEENDIAN;
    tif->tif_header.tiff_version = TIFF_VERSION;
    tif->tif_header.tiff_diroff = 0;    /* filled in later */
    if (!WriteOK(fd, &tif->tif_header, sizeof (TIFFHeader))) {
        TIFFError(name, "Error writing TIFF header");
        goto bad;
    }
    /*
     * Set up the byte-order handling.
     */
    TIFFInitOrder(tif, tif->tif_header.tiff_magic, bigendian);
    /*
     * Set up default directory.
     */
    if (!TIFFDefaultDirectory(tif))
        goto bad;
    tif->tif_diroff = 0;
    return (tif);
}
```

If the file has a header, more work lies ahead. We check the header fields to be sure we have a plausible TIFF file. Assuming we do, we read in the first TIFF directory, note that no user buffer has been supplied (TIFF_MYBUFFER), and set the pointers used for buffering image data to mark that no data is present, and finally read in the first IFD using TIFFReadDirectory( ).

```
    /*
     * Set up the byte-order handling.
     */
    if (tif->tif_header.tiff_magic != TIFF_BIGENDIAN &&
        tif->tif_header.tiff_magic != TIFF_LITTLEENDIAN) {
        TIFFError(name,  "Not a TIFF file,
                  bad magic number %d (0x%x)",
            tif->tif_header.tiff_magic,
            tif->tif_header.tiff_magic);
        goto bad;
    }
    TIFFInitOrder(tif, tif->tif_header.tiff_magic, bigendian);
    /*
     * Swap header if required.
     */
    if (tif->tif_flags & TIFF_SWAB) {
        TIFFSwabShort(&tif->tif_header.tiff_version);
        TIFFSwabLong(&tif->tif_header.tiff_diroff);
    }
    /*
     * Now check version (if needed, it's been byte-swapped).
```

```
     * Note that this isn't actually a version number, it's a
     * magic number that doesn't change (stupid).
     */
    if (tif->tif_header.tiff_version != TIFF_VERSION) {
        TIFFError(name,
            "Not a TIFF file, bad version number %d (0x%x)",
            tif->tif_header.tiff_version,
            tif->tif_header.tiff_version);
        goto bad;
    }
    tif->tif_flags |= TIFF_MYBUFFER;
    tif->tif_rawcp = tif->tif_rawdata = 0;
    tif->tif_rawdatasize = 0;
    /*
     * Set up initial directory.
     */
    switch (mode[0]) {
    case 'r':
        tif->tif_nextdiroff = tif->tif_header.tiff_diroff;
        if (TIFFReadDirectory(tif)) {
            tif->tif_rawcc = -1;
            tif->tif_flags |= TIFF_BUFFERSETUP;
            return (tif);
        }
        break;
    case 'a':
        /* code for appending to existing files omitted */
    }
bad:
    tif->tif_mode = O_RDONLY;    /* XXX avoid flush */
    TIFFClose(tif);
    return ((TIFF *)0);
bad2:
    (void) close(fd);
    return ((TIFF *)0);
}

TIFFScanlineSize(tif)
    TIFF *tif;
{
    TIFFDirectory *td = &tif->tif_dir;
    long scanline;

    scanline = td->td_bitspersample * td->td_imagewidth;
    if (td->td_planarconfig == PLANARCONFIG_CONTIG)
        scanline *= td->td_samplesperpixel;
    return (howmany(scanline, 8));
}
```

Since application programs can't see the internal fields of a TIFF structure, here we define some routines to return miscellaneous fields.

```
/*
 * Query functions to access private data.
 */

/*
 * Return open file's name.
 */
char *
TIFFFileName(tif)
    TIFF *tif;
{
    return (tif->tif_name);
}

/*
 * Return open file's I/O descriptor.
 */
int
TIFFFileno(tif)
    TIFF *tif;
{
    return (tif->tif_fd);
}

/*
 * Return read/write mode.
 */
int
TIFFGetMode(tif)
    TIFF *tif;
{
    return (tif->tif_mode);
}

/*
 * Return non-zero if file is organized in
 * tiles; zero if organized as strips.
 */
int
TIFFIsTiled(tif)
    TIFF *tif;
{
    return (isTiled(tif));
}
```

```
/*
 * Return current row being read/written.
 */
long
TIFFCurrentRow(tif)
    TIFF *tif;
{
    return (tif->tif_row);
}

/*
 * Return index of the current directory.
 */
int
TIFFCurrentDirectory(tif)
    TIFF *tif;
{
    return (tif->tif_curdir);
}

/*
 * Return current strip.
 */
int
TIFFCurrentStrip(tif)
    TIFF *tif;
{
    return (tif->tif_curstrip);
}

/*
 * Return current tile.
 */
int
TIFFCurrentTile(tif)
    TIFF *tif;
{
    return (tif->tif_curtile);
}
```

## Closing TIFF Files

Closing a TIFF file is straightforward. First we flush any buffered image data, using the routine shown below. If the image encoding scheme in use has a cleanup routine, we call it. (We discuss this in more detail below.) Finally we free the directory and, if one is present, the buffer, close the file, and free the TIFF structure.

```
/* tif_close.c
 * TIFF Library.
 */
#include "tiffioP.h"

#if USE_PROTOTYPES
extern  int TIFFFreeDirectory(TIFF*);
#else
extern  int TIFFFreeDirectory();
#endif

void
TIFFClose(tif)
    TIFF *tif;
{
    if (tif->tif_mode != O_RDONLY)
        /*
         * Flush buffered data and directory (if dirty).
         */
        TIFFFlush(tif);
    if (tif->tif_cleanup)
        (*tif->tif_cleanup)(tif);
    TIFFFreeDirectory(tif);
    if (tif->tif_rawdata && (tif->tif_flags&TIFF_MYBUFFER))
        free(tif->tif_rawdata);
    (void) close(tif->tif_fd);
    free((char *)tif);
}
```

The usual time to flush data to the disk is when the file is being closed, as we just saw, but an application can, if it wants, flush at any other time as well.

Assuming the file is not open read-only, TIFFFlush( ) flushes any buffered image data (shown next), and if the in-memory directory has been changed, writes the IFD.

TIFFFlushData( ) checks whether we've written anything at all. If so, and there is a post-encoding routine for the encoding technique, it calls it, followed by a call to TIFFFlushData1( ), which actually writes the data to the file. We'll see this latter routine in an upcoming section on writing image data.

```
/* tif_flush.c
 * TIFF Library.
 */
#include "tiffioP.h"

TIFFFlush(tif)
    TIFF *tif;
{
```

```
    if (tif->tif_mode != O_RDONLY) {
        if (!TIFFFlushData(tif))
            return (0);
        if ((tif->tif_flags & TIFF_DIRTYDIRECT) &&
            !TIFFWriteDirectory(tif))
            return (0);
    }
    return (1);
}

/*
 * Flush buffered data to the file.
 */
TIFFFlushData(tif)
    TIFF *tif;
{
    if ((tif->tif_flags & TIFF_BEENWRITING) == 0)
        return (0);
    if (tif->tif_flags & TIFF_POSTENCODE) {
        tif->tif_flags &= ~TIFF_POSTENCODE;
        if (tif->tif_postencode && !(*tif->tif_postencode)(tif))
            return (0);
    }
    return (TIFFFlushData1(tif));
}
```

## 18.5 COMPRESSION AND ENCODING TECHNIQUES

TIFF supports about a dozen different compression techniques, far more than any other image format. The TIFF library handles this embarrassment of riches in an object-oriented way. Each TIFF structure has a set of function pointers to the compression-specific routines. Whenever the library needs to perform a compression function on a TIFF file, it calls the routine pointed to by that file's TIFF structure. This lets the library handle multiple files with different compression formats at the same time, since their TIFF structures can point to different sets of routines, and it also makes it possible to add a new compression technique without having to modify every routine that calls any compression code; the only thing that has to be changed is the array CompressionSchemes, which lists the compression methods.

The functions that may be defined by each compression method are:

**tif_predecode** Called before doing any image decoding.

**tif_preencode** Called before doing any image encoding.

**tif_postencode** Called after doing any image encoding.

**tif_decoderow** Called to decode a row of pixels.

**tif_encoderow** Called to encode a row of pixels.

**tif_decodestrip** Called to decode an entire strip of pixels.

**tif_encodestrip** Called to encode an entire strip of pixels.

**tif_decodetile** Called to decode a tile of pixels.

**tif_encodetile** Called to encode a tile of pixels.

**tif_close** Called when the file is closed.

**tif_seek** Called to position to a particular row of the image.

**tif_cleanup** Called when the file is closed or when the application switches to another compression method.

The application or the IFD reader sets the compression method using TIFFSetCompressionScheme( ). It looks up the compression method in the table Compression Schemes, using findScheme( ), sets all of the compression method pointers to a default, and then calls the init routine for the method. The defaults are to do nothing for pre-decode, pre-encode, post-encode, close, seek, and cleanup, and to call a routine that prints an error and fails for the other routines. The method's init routine installs pointers to the actual routines for the method, replacing some or all of the default pointers.

```
/* tif_compress.c
 * TIFF Library
 *
 * Compression Scheme Configuration Support.
 */
#include "tiffioP.h"

#if USE_PROTOTYPES
extern  int TIFFInitDumpMode(TIFF*);
#ifdef PACKBITS_SUPPORT
extern  int TIFFInitPackBits(TIFF*);
#endif
#ifdef CCITT_SUPPORT
extern  int TIFFInitCCITTRLE(TIFF*), TIFFInitCCITTRLEW(TIFF*);
extern  int TIFFInitCCITTFax3(TIFF*), TIFFInitCCITTFax4(TIFF*);
#endif
#ifdef THUNDER_SUPPORT
extern  int TIFFInitThunderScan(TIFF*);
#endif
#ifdef NEXT_SUPPORT
extern  int TIFFInitNeXT(TIFF*);
#endif
#ifdef LZW_SUPPORT
extern  int TIFFInitLZW(TIFF*);
```

```
#endif
#ifdef JPEG_SUPPORT
extern  int TIFFInitJPEG(TIFF*);
#endif
#else
extern  int TIFFInitDumpMode();
#ifdef PACKBITS_SUPPORT
extern  int TIFFInitPackBits();
#endif
#ifdef CCITT_SUPPORT
extern  int TIFFInitCCITTRLE(), TIFFInitCCITTRLEW();
extern  int TIFFInitCCITTFax3(), TIFFInitCCITTFax4();
#endif
#ifdef THUNDER_SUPPORT
extern  int TIFFInitThunderScan();
#endif
#ifdef NEXT_SUPPORT
extern  int TIFFInitNeXT();
#endif
#ifdef LZW_SUPPORT
extern  int TIFFInitLZW();
#endif
#ifdef JPEG_SUPPORT
extern  int TIFFInitJPEG();
#endif
#endif

struct cscheme {
    char*   name;
    int scheme;
    int (*init)();
};
static const struct cscheme CompressionSchemes[] = {
    { "Null",       COMPRESSION_NONE,   TIFFInitDumpMode },
#ifdef LZW_SUPPORT
    { "LZW",        COMPRESSION_LZW,    TIFFInitLZW },
#endif
#ifdef PACKBITS_SUPPORT
    { "PackBits",   COMPRESSION_PACKBITS,   TIFFInitPackBits },
#endif
#ifdef THUNDER_SUPPORT
    { "ThunderScan",
                    COMPRESSION_THUNDERSCAN,TIFFInitThunderScan },
#endif
#ifdef NEXT_SUPPORT
    { "NeXT",       COMPRESSION_NEXT,   TIFFInitNeXT },
#endif
#ifdef JPEG_SUPPORT
    { "JPEG",       COMPRESSION_JPEG,   TIFFInitJPEG },
```

```
#endif
#ifdef CCITT_SUPPORT
    { "CCITT RLE",  COMPRESSION_CCITTRLE,   TIFFInitCCITTRLE },
    { "CCITT RLE/W",    COMPRESSION_CCITTRLEW,  TIFFInitCCITTRLEW },
    { "CCITT Group3",   COMPRESSION_CCITTFAX3,  TIFFInitCCITTFax3 },
    { "CCITT Group4",   COMPRESSION_CCITTFAX4,  TIFFInitCCITTFax4 },
#endif
};
#define NSCHEMES (sizeof (CompressionSchemes) /
        sizeof (CompressionSchemes[0]))

static struct cscheme const *
findScheme(scheme)
    int scheme;
{
    register struct cscheme const *c;

    for (c = CompressionSchemes; c <
            &CompressionSchemes[NSCHEMES]; c++)
        if (c->scheme == scheme)
            return (c);
    return ((struct cscheme const *)0);
}
```

Here are the default encoding and decoding routines, installed unless the init routine for the actual encoding method overrides them.

```
static int
TIFFNoEncode(tif, method)
    TIFF *tif;
    char *method;
{
    struct cscheme const *c = findScheme(tif->
    tif_dir.td_compression);
    TIFFError(tif->tif_name,
        "%s %s encoding is not implemented", c->name, method);
    return (-1);
}

int
TIFFNoRowEncode(tif, pp, cc, s)
    TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
    return (TIFFNoEncode(tif, "scanline"));
}
```

```
int
TIFFNoStripEncode(tif, pp, cc, s)
    TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
    return (TIFFNoEncode(tif, "strip"));
}

int
TIFFNoTileEncode(tif, pp, cc, s)
    TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
    return (TIFFNoEncode(tif, "tile"));
}

int
TIFFNoDecode(tif, method)
    TIFF *tif;
    char *method;
{
    struct cscheme const *c = findScheme(tif->
    tif_dir.td_compression);
    TIFFError(tif->tif_name,
        "%s %s decoding is not implemented", c->name, method);
    return (-1);
}

int
TIFFNoRowDecode(tif, pp, cc, s)
    TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
    return (TIFFNoDecode(tif, "scanline"));
}

int
TIFFNoStripDecode(tif, pp, cc, s)
    TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
```

```
    return (TIFFNoDecode(tif, "strip"));
}

int
TIFFNoTileDecode(tif, pp, cc, s)
    TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
    return (TIFFNoDecode(tif, "tile"));
}

TIFFSetCompressionScheme(tif, scheme)
    TIFF *tif;
    int scheme;
{
    struct cscheme const *c = findScheme(scheme);

    if (!c) {
        TIFFError(tif->tif_name,
            "Unknown data compression algorithm %u (0x%x)",
            scheme, scheme);
        return (0);
    }
    tif->tif_predecode = NULL;
    tif->tif_decoderow = TIFFNoRowDecode;
    tif->tif_decodestrip = TIFFNoStripDecode;
    tif->tif_decodetile = TIFFNoTileDecode;
    tif->tif_preencode = NULL;
    tif->tif_postencode = NULL;
    tif->tif_encoderow = TIFFNoRowEncode;
    tif->tif_encodestrip = TIFFNoStripEncode;
    tif->tif_encodetile = TIFFNoTileEncode;
    tif->tif_close = NULL;
    tif->tif_seek = NULL;
    tif->tif_cleanup = NULL;
    tif->tif_flags &= ~TIFF_NOBITREV;
    tif->tif_options = 0;
    return ((*c->init)(tif));
}
```

## No Compression

The simplest and most common compression scheme is to do no compression at all. The "null" compression scheme merely copies the pixels directly in and out of the file. This scheme's advantages are that it is simple, fast, and complete. (It is the only scheme where you can directly access any pixel in the image.) The disadvantage, of course, is that it doesn't save any space.

The initialization routine installs three method routines: DumpModeDecode( ), which handles all of the "decompression," copying from the file to the application; DumpModeEncode( ), which handles all of the "compression," copying from the application to the file; and DumpModeSeek( ), which handles seeking to any particular row in the image.

```
/* tif_dumpmode.c
 * TIFF Library.
 *
 * "Null" Compression Algorithm Support.
 */

#include "tiffioP.h"
#include <stdio.h>
#include <assert.h>

#if USE_PROTOTYPES
static  int DumpModeEncode(TIFF *, u_char *, int, u_int);
static  int DumpModeDecode(TIFF *, u_char *, int, u_int);
static  int DumpModeSeek(TIFF *, int);
#else
static  int DumpModeEncode(), DumpModeDecode(), DumpModeSeek();
#endif

/*
 * Initialize dump mode.
 */

TIFFInitDumpMode(tif)
    register TIFF *tif;
{
    tif->tif_decoderow = DumpModeDecode;
    tif->tif_decodestrip = DumpModeDecode;
    tif->tif_decodetile = DumpModeDecode;
    tif->tif_encoderow = DumpModeEncode;
    tif->tif_encodestrip = DumpModeEncode;
    tif->tif_encodetile = DumpModeEncode;
    tif->tif_seek = DumpModeSeek;
    return (1);
}
```

All of the encode routines take the same set of arguments. (They have to, since they are called interchangeably depending on the compression method in use.) The arguments are tif, the TIFF structure; pp, the buffer from which to take the pixels; cc, the number of bytes in the buffer; and s, the sample number.

Several fields in the TIFF structure keep track of the buffer used for reading and writing image data:

**tif_scanlinesize** The size of a row of uncompressed pixels, in bytes.

***tif_rawdata** The buffer used for reading or writing compressed data to and from the file.

**tif_rawdatasize** The size of that buffer.

***tif_rawcp** The current spot in the raw buffer to read or write next.

**tif_rawcc** When writing, the number of valid bytes stored so far in the raw buffer.

**tif_data** A private field for the use of the encoding and decoding routines. The no-compression routines don't use it. Most of the other methods use it to store a pointer to a structure of method-specific data.

The raw buffer is usually allocated by the library but may be provided by the application by calling TIFFReadBufferSetup( ), in case the application wants to use a larger buffer than otherwise would be allocated.

The no-compression encoded basically copies pixel data from the user buffer pointed to by pp, size cc, into the file buffer described above.

```
/*
 * Encode a hunk of pixels.
 */
static int
DumpModeEncode(tif, pp, cc, s)
    register TIFF *tif;
    u_char *pp;
    int cc;
    u_int s;
{
    /*
     * This may be overzealous but avoids having to
     * worry about byte alignment for the (potential)
     * byte-swapping work below.
     */
```

If the file buffer is full, it calls TIFFFlushData1( ), shown later, to flush the buffer to the file and reset the pointers.

```
    if (tif->tif_rawcc + cc > tif->tif_rawdatasize)
        if (!TIFFFlushData1(tif))
            return (-1);
```

If the image uses multibyte components (uncommon, but possible in high-resolution images), we have to swap the bytes if the file byte order is different from the host byte order. Other than that, we just copy the bytes with bcopy( ) and flush the data to the file as needed.

```
    while (cc > 0) {
        int n;
        if ((n = cc) > tif->tif_rawdatasize)
            n = tif->tif_rawdatasize;
        bcopy(pp, tif->tif_rawcp, n);
        if (tif->tif_flags & TIFF_SWAB) {
            switch (tif->tif_dir.td_bitspersample) {
            case 16:
                assert((n & 3) == 0);
                TIFFSwabArrayOfShort((u_short *)tif->tif_rawcp,
                    n/2);
                break;
            case 32:
                assert((n & 15) == 0);
                TIFFSwabArrayOfLong((u_long *)tif->tif_rawcp,
                    n/4);
                break;
            }
        }
        tif->tif_rawcp += n;
        tif->tif_rawcc += n;
        pp += n;
        cc -= n;
        if (tif->tif_rawcc >= tif->tif_rawdatasize &&
            !TIFFFlushData1(tif))
            return (-1);
    }
    return (1);
}
```

"Decoding" data in no-compression mode is simpler than encoding because then we can assume the caller has already read the file data in. The only work is byte swapping.

```
/*
 * Decode a hunk of pixels. */
static int
DumpModeDecode(tif, buf, cc, s)
    register TIFF *tif;
    u_char *buf;
    int cc;
    u_int s;
{
    if (tif->tif_rawcc < cc) {
        TIFFError(tif->tif_name,
            "DumpModeDecode: Not enough data for scanline %d",
            tif->tif_row);
        return (0);
    }
```

```
/* Avoid copy if client has set up raw
 * data buffer to avoid extra copy.
 */
if (tif->tif_rawcp != (char*) buf)
    bcopy(tif->tif_rawcp, buf, cc);
if (tif->tif_flags & TIFF_SWAB) {
    switch (tif->tif_dir.td_bitspersample) {
    case 16:
        assert((cc & 3) == 0);
        TIFFSwabArrayOfShort((u_short *)buf, cc/2);
        break;
    case 32:
        assert((cc & 15) == 0);
        TIFFSwabArrayOfLong((u_long *)buf, cc/4);
        break;
    }
}
tif->tif_rawcp += cc;
tif->tif_rawcc -= cc;
return (1);
}
```

Seeking in an uncompressed file is simple, since all rows are the same size. To move N rows forward or backward, we simply adjust the pointer N times the size of a scan line.

Other library code, shown later, takes care of seeking from one strip to another.

```
/*
 * Seek forward nrows in the current strip.
 */
static int
DumpModeSeek(tif, nrows)
    register TIFF *tif;
    int nrows;
{
    tif->tif_rawcp += nrows * tif->tif_scanlinesize;
    tif->tif_rawcc -= nrows * tif->tif_scanlinesize;
    return (1);
}
```

## PackBits Compression

The next most common compression technique is the familiar PackBits, which we have already seen in MacPaint and ILBM images.

There are more routines defined here—a pre-encode routine, which is called before any encoding happens; and an encode-chunk routine, which is separate from the regular encode routine. The former encodes an entire strip, while the latter handles a row at a time.

```
/*
 * tif_packbits.c
 * TIFF Library.
 *
 * PackBits Compression Algorithm Support
 */
#include "tiffioP.h"
#include <stdio.h>
#include <assert.h>

#if USE_PROTOTYPES
static  int PackBitsPreEncode(TIFF *);
static  int PackBitsEncode(TIFF *, u_char *, int, u_int);
static  int PackBitsEncodeChunk(TIFF *, u_char *, int, u_int);
static  int PackBitsDecode(TIFF *, u_char *, int, u_int);
#else
/*
 * nonprototype declarations */
#endif

TIFFInitPackBits(tif)
    TIFF *tif;
{
    tif->tif_decoderow = PackBitsDecode;
    tif->tif_decodestrip = PackBitsDecode;
    tif->tif_decodetile = PackBitsDecode;
    tif->tif_preencode = PackBitsPreEncode;
    tif->tif_encoderow = PackBitsEncode;
    tif->tif_encodestrip = PackBitsEncodeChunk;
    tif->tif_encodetile = PackBitsEncodeChunk;
    return (1);
}
```

The pre-encode routine calculates the size of an uncompressed row (or, in tiled data, the size of a piece of a row in a tile) in bytes and puts it in tif_data.

```
static int
PackBitsPreEncode(tif)
    TIFF *tif;
{
    /*
     * Calculate the scan-line/tile-width size in bytes.
     */
    if (isTiled(tif))
        tif->tif_data = (char *) TIFFTileRowSize(tif);
    else
        tif->tif_data = (char *) TIFFScanlineSize(tif);
    return (1);
}
```

PackBitsEncodeChunk( ) handles encoding an entire strip or tile. As the comments note, it encodes a row at a time to be sure that a single encoded item doesn't span rows.

```
/*
 * Encode a rectangular chunk of pixels.  We break it up
 * into row-sized pieces to ensure that encoded runs do
 * not span rows.  Otherwise, there can be problems with
 * the decoder if data is read, for example, by scan lines
 * when it was encoded by strips.
 */
static int
PackBitsEncodeChunk(tif, bp, cc, s)
    TIFF *tif;
    u_char *bp;
    int cc;
    u_int s;
{
    int rowsize = (int) tif->tif_data;

    assert(rowsize > 0);
    while (cc > 0) {
        if (PackBitsEncode(tif, bp, rowsize, s) < 0)
            return (-1);
        bp += rowsize;
        cc -= rowsize;
    }
    return (1);
}
```

PackBitsEncode( ) does the real work of encoding image data. It takes the same arguments as DumpModeEncode( ). (It has to—it's called from the same places.)

The encoder can be in four different states:

**BASE** It just started, or it just wrote out a maximum-length literal group.

**LITERAL** It is writing a literal group that can be made longer.

**RUN** It just wrote a run of repeated bytes.

**LITERAL_RUN** Special case: it just wrote a literal string followed by a run. If the next thing is a literal, and the run was only 2 bytes, it is more compact to convert the whole thing to one long literal.

```
/*
 * Encode a run of pixels.
 */
static int
```

```
PackBitsEncode(tif, bp, cc, s)
    TIFF *tif;
    u_char *bp;
    register int cc;
    u_int s;
{
    register char *op,          /* current output pointer */
                 *lastliteral;
                        /* latest literal PackBits descriptor */
    register int n, b;
    enum { BASE, LITERAL, RUN, LITERAL_RUN } state;
    char *ep;          /* end of output buffer */
    int slop;

    op = tif->tif_rawcp;
    ep = tif->tif_rawdata + tif->tif_rawdatasize;
    state = BASE;
    lastliteral = 0;

    while (cc > 0) { /* iterate until entire row is compressed */
        /*
         * Find the longest string of identical bytes.
         */
        b = *bp++, cc--, n = 1;
        for (; cc > 0 && b == *bp; cc--, bp++)
            n++;
    again:

        if (op + 2 >= ep) {     /* ensure space for new data */
            /*
             * If the output buffer is nearly full, flush it
             * but keep any literal chunk we are building.
             * Be careful about writing the last
             * literal.  Must write up to that point
             * and then copy the remainder to the
             * front of the buffer.
             */
            if (state == LITERAL || state == LITERAL_RUN) {
                slop = op - lastliteral;
                tif->tif_rawcc += lastliteral - tif->tif_rawcp;
                if (!TIFFFlushData1(tif))
                    return (-1);
                op = tif->tif_rawcp;
                for (; slop-- > 0; *op++ = *lastliteral++)
                    ;
                lastliteral = tif->tif_rawcp;
            } else {
                tif->tif_rawcc += op - tif->tif_rawcp;
                if (!TIFFFlushData1(tif))
```

```
                return (-1);
            op = tif->tif_rawcp;
        }
    }
```

We know there is space in the output buffer, and we know how long the next run can be. The particular action depends on which of the four states we are in.

```
    switch (state) {
    case BASE:          /* initial state, set run/literal */
        if (n > 1) {        /* repeated data, write a run */
            state = RUN;
            if (n > 128) {
                    /* PackBits has a max repeat of 128 */
                *op++ = -127;
                *op++ = b;
                n -= 128;
                goto again;
            }
            *op++ = -(n-1);
            *op++ = b;
        } else {
            lastliteral = op;
                        /* start a literal string here */
            *op++ = 0;
            *op++ = b;
            state = LITERAL;
        }
        break;
    case LITERAL:       /* last object was literal string */
        if (n > 1) {
            state = LITERAL_RUN;
            if (n > 128) {
                *op++ = -127;
                *op++ = b;
                n -= 128;
                goto again;
            }
            *op++ = -(n-1);     /* encode run */
            *op++ = b;
        } else {            /* extend literal */
            if (++(*lastliteral) == 127)
                state = BASE;
            *op++ = b;
        }
        break;
    case RUN:       /* last object was run */
        if (n > 1) {
            if (n > 128) {
```

```
                        *op++ = -127;
                        *op++ = b;
                        n -= 128;
                        goto again;
                    }
                    *op++ = -(n-1);
                    *op++ = b;
                } else {
                    lastliteral = op;
                    *op++ = 0;
                    *op++ = b;
                    state = LITERAL;
                }
                break;
            case LITERAL_RUN:   /* literal followed by a run */
                /*
                 * Check to see if previous run should
                 * be converted to a literal, in which
                 * case we convert literal-run-literal
                 * to a single literal.
                    * We do so if the run was only two long and
                    * there is room to increase the previous literal
                 */
                if (n == 1 && op[-2] == (char)-1 &&
                    *lastliteral < 126) {
                    state = (((*lastliteral) += 2) == 127 ?
                        BASE : LITERAL);
                    op[-2] = op[-1];    /* replicate */
                } else
                    state = RUN;
                goto again;
            }
        }
        tif->tif_rawcc += op - tif->tif_rawcp;
                                /* update output buffer  ptr */
        tif->tif_rawcp = op;
        return (1);
}
```

Decoding PackBits is easier than encoding since the work is entirely dictated by the PackBits control bytes in the buffer. We read and expand the compressed data until the output buffer is full, in which case we're done, or we run out of compressed data, in which case the file must have been corrupted.

```
static int
PackBitsDecode(tif, op, occ, s)
    TIFF *tif;
    register u_char *op;
    register int occ;
```

```
    u_int s;
{
    register char *bp;
    register int n, b;
    register int cc;

    bp = tif->tif_rawcp; cc = tif->tif_rawcc;
    while (cc > 0 && occ > 0) {
        n = (int) *bp++;
        /*
         * Watch out for compilers that
         * don't sign extend chars...
         */
        if (n >= 128)
            n -= 256;
        if (n < 0) {        /* replicate next byte -n+1 times */
            cc--;
            if (n == -128)  /* nop */
                continue;
            n = -n + 1;
            occ -= n;
            for (b = *bp++; n-- > 0;)
                *op++ = b;
        } else {            /* copy next n+1 bytes literally */
            bcopy(bp, op, ++n);
            op += n; occ -= n;
            bp += n; cc -= n;
        }
    }
    tif->tif_rawcp = bp;
    tif->tif_rawcc = cc;
    if (occ > 0) {
        TIFFError(tif->tif_name,
            "PackBitsDecode: Not enough data for scanline %d",
            tif->tif_row);
        return (0);
    }
    /* check for buffer overruns? */
    return (1);
}
```

## Other Compression Methods

The TIFF library code on the companion disk supports many other compression methods with code too long to show here. They include:

**CCITT RLE encoding** An encoding for single-bit images that uses a fixed table of variable-length codes to encode runs of black or white bits in the image. Very effective for scanned black-and-white images, e.g., faxes. This

is basically Group 3 fax compression but without some extra codes in the data used by actual fax machines.

**Group 3 fax** Same as above, but with all of the fax tags.

**Group 4 fax** An improved version of Group 3 that does two-dimensional encoding, which means that it takes advantage of similarities from one row to the next to improve the compression. Also, a rarely used uncompressed submode handles images that don't compress well using the regular rules.

**LZW** The same dictionary encoding used in GIF files. (It's not quite the same; because of bugs in early TIFF implementations, a few details differ.)

**JPEG** JPEG compression, which is extremely effective on grayscale and color scanned photographic images. See Chapter 19.

The library also handles some common nonstandard compression techniques:

**Thunderscan** A proprietary scheme used by Thunderscan scanners, which handles literal strings, repeated pixels, and also *delta* coding, which stores the differences between adjacent pixels when the values change slowly.

**NeXT** A specialized RLE technique effective on 2-bit per pixel grayscale images found on many NeXT systems.

**Pixar RLE** An RLE format from Pixar specialized for encoding four-component ARGB images.

**Silicon Graphics RLE** A variation on PackBits that compresses 16-bit data, particularly effective on images with 16-bit components.

## 18.6 READING TIFF DATA

The TIFF library has several different sets of reading routines that let applications read by row, by strip, or by tile, with or without handling the image compression. We look at the most straightforward set of routines, the ones that read and write a row of pixels at a time. The primary entry point is TIFFReadScanline( ), which takes four arguments:

- The TIFF structure returned by TIFFOpen( ).
- The buffer into which to put the decoded row of pixels.
- The row number.
- If the image is full colors and stores the components separately, which set of components (known in TIFF-ese as which sample) to return.

If pixels are 1, 2, or 4 bits apiece, they are packed into bytes in the obvious way.

Reading image data at the top level involves seeking to the appropriate place in the file then calling the compression method's decode routine.

```
/*
 * tif_read.c
 * TIFF Library.
 * Scanline-oriented Read Support
 */
#include "tiffioP.h"

#if USE_PROTOTYPES
static  TIFFSeek(TIFF *, u_int, u_int);
static  int TIFFReadRawStrip1(TIFF *, u_int, u_char *,
        u_int, char []);
static  TIFFFillStrip(TIFF *, u_int);
static  TIFFStartStrip(TIFF *, u_int);
static  TIFFCheckRead(TIFF *, int);
#else
/* nonprototype version */
#endif

/*VARARGS3*/
TIFFReadScanline(tif, buf, row, sample)
    register TIFF *tif;
    u_char *buf;
    u_int row, sample;
{
    int e;

    if (!TIFFCheckRead(tif, 0)) /* open for reading? */
        return (-1);
    if (e = TIFFSeek(tif, row, sample)) {
        /*
         * Decompress desired row into user buffer.
         */
        e = (*tif->tif_decoderow)(tif, buf, tif->tif_scanlinesize,
            sample);
        tif->tif_row++;
    }
    return (e ? 1 : -1);
}

/*
 * Make sure we can read this file and that we're not doing the
 * wrong kind of read
 */
```

```
static
TIFFCheckRead(tif, tiles)
    TIFF *tif;
    int tiles;
{
    if (tif->tif_mode == O_WRONLY) {
        TIFFError(tif->tif_name, "File not open for reading");
        return (0);
    }
    if (tiles ^ isTiled(tif)) {
        TIFFError(tif->tif_name, tiles ?
            "Cannot read tiles from a stripped image" :
            "Cannot read scanlines from a tiled image");
        return (0);
    }
    return (1);
}
```

Seeking to an arbitrary file is complicated by the possibility that the image may be stored in strips or tiles or as separate components.

If the components are stored separately, we essentially have a complete image for each component, so we skip ahead to the Nth image to read the Nth component. In the more common case, there is only one component, or the components are stored together. We then compute the strip number directly from the row number, since all strips contain the same number of rows.

```
/*
 * Seek to a random row+sample in a file.
 */
static
/*VARARGS2*/
TIFFSeek(tif, row, sample)
    register TIFF *tif;
    u_int row, sample;
{
    register TIFFDirectory *td = &tif->tif_dir;
    int strip;

    if (row >= td->td_imagelength) {    /* out of range */
        TIFFError(tif->tif_name, "%d: Row out of range, max %d",
            row, td->td_imagelength);
        return (0);
    }
    if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
        if (sample >= td->td_samplesperpixel) {
            TIFFError(tif->tif_name,
                "%d: Sample out of range, max %d",
```

```
                    sample, td->td_samplesperpixel);
                return (0);
            }
            strip = sample*td->td_stripsperimage +
                                                row/td->td_rowsperstrip;
        } else
            strip = row / td->td_rowsperstrip;
```

If the strip isn't already in memory, we read it with TIFFStartStrip( ). Then, if we aren't already at the correct row, we use the compression method's seek routine, if one exists, to go to the correct row. Some compression techniques don't allow seeking from row to row, since the row boundaries aren't necessarily at easily describable places in the compressed data.

```
    if (strip != tif->tif_curstrip) {
                                    /* different strip, refill */
        if (!TIFFFillStrip(tif, strip))
            return (0);
    } else if (row < tif->tif_row) {

        /*
         * Moving backward within the same strip: back up
         * to the start and then decode forward (below).
         *
         * NB: If you're planning on lots of random access within
         * a strip, it's better to just read and decode the entire
         * strip and then access the decoded data in a random
         * fashion.
         */

        if (!TIFFStartStrip(tif, strip))
            return (0);
    }
    if (row != tif->tif_row) {
        if (tif->tif_seek) {
            /*
             * Seek forward to the desired row.
             */
            if (!(*tif->tif_seek)(tif, row - tif->tif_row))
                return (0);
            tif->tif_row = row;
        } else {
            TIFFError(tif->tif_name,
            "Compression algorithm does not support random access");
            return (0);
        }
    }
    return (1);
}
```

Once we have seeked to the correct strip, we have to read in the strip using TIFFFillStrip( ). We check to see, based on the strip's byte count, if the buffer is big enough to hold this strip. (The byte count should have been read from the IFD, or if it wasn't present, guessed with EstimateStripByteCounts( ) above.) If the buffer is too small, we try to get a bigger one, using TIFFReadBufferSetup( ). Then we read the actual data with TIFFReadRawStrip1( ). If the file uses a different bit order than we do, we use TIFFReverseBits to flip the bits around. Finally, TIFFStartStrip( ) sets up the pointers to start decoding the strip.

```
/*
 * Read the specified strip and set up for decoding.
 * The data buffer is expanded, as necessary, to
 * hold the strip's data.
 */
static
TIFFFillStrip(tif, strip)
    TIFF *tif;
    u_int strip;
{
    static char module[] = "TIFFFillStrip";
    TIFFDirectory *td = &tif->tif_dir;
    u_long bytecount;

    bytecount = td->td_stripbytecount[strip];
        /*
         * Expand raw data buffer, if needed, to
         * hold data strip coming from file
         * (perhaps should set upper bound on
         *  the size of a buffer we'll use?).
         */
        if (bytecount > tif->tif_rawdatasize) {
            tif->tif_curstrip = -1;     /* unknown state */
            if ((tif->tif_flags & TIFF_MYBUFFER) == 0) {
                TIFFError(module,
                "%s: Data buffer too small to hold strip %d",
                    tif->tif_name, strip);
                return (0);
            }
            if (!TIFFReadBufferSetup(tif, 0,
                roundup(bytecount, 1024)))
                return (0);
    }
    if (TIFFReadRawStrip1(tif, strip,
        (u_char *)tif->tif_rawdata,
        bytecount, module) != bytecount)
        return (0);
    if (td->td_fillorder != tif->tif_fillorder &&
        (tif->tif_flags & TIFF_NOBITREV) == 0)
        TIFFReverseBits((u_char *)tif->tif_rawdata,
```

```
                bytecount);
        return (TIFFStartStrip(tif, strip));
}
```

TIFFReadBufferSetup( ) gives back any existing buffer and then either installs an application-provided buffer (if it is called with a non-null bp) or allocates a new buffer of appropriate size.

```
/*
 * Set up the raw data buffer in preparation for
 * reading a strip of raw data.  If the buffer
 * is specified as zero, then a buffer of appropriate
 * size is allocated by the library.  Otherwise,
 * the client must guarantee that the buffer is
 * large enough to hold any individual strip of
 * raw data.
 */
int
TIFFReadBufferSetup(tif, bp, size)
    TIFF *tif;
    char *bp;
    u_int size;
{
    static char module[] = "TIFFReadBufferSetup";

    if (tif->tif_rawdata) {
        if (tif->tif_flags & TIFF_MYBUFFER)
            free(tif->tif_rawdata);
        tif->tif_rawdata = NULL;
    }
    if (bp) {
        tif->tif_rawdatasize = size;
        tif->tif_rawdata = bp;
        tif->tif_flags &= ~TIFF_MYBUFFER;
    } else {
        tif->tif_rawdatasize = roundup(size, 1024);
        tif->tif_rawdata = malloc(tif->tif_rawdatasize);
        tif->tif_flags |= TIFF_MYBUFFER;
    }
    if (tif->tif_rawdata == NULL) {
        TIFFError(module,
            "%s: No space for data buffer at scanline %d",
            tif->tif_name, tif->tif_row);
        tif->tif_rawdatasize = 0;
        return (0);
    }
    return (1);
}
```

Finally, we get around to reading the data. TIFFReadRawStrip1( ) uses SeekOK( ), a wrapper around the lseek( ) system call, to go to the correct place in the file, and ReadOK( ), a wrapper around the read( ) system call, to read the data.

```
static int
TIFFReadRawStrip1(tif, strip, buf, size, module)
    TIFF *tif;
    u_int strip;
    u_char *buf;
    u_int size;
    char module[];
{
    TIFFDirectory *td = &tif->tif_dir;

        if (!SeekOK(tif->tif_fd, td->td_stripoffset[strip])) {
            TIFFError(module,
                "%s: Seek error at scanline %d, strip %d",
                tif->tif_name, tif->tif_row, strip);
            return (-1);
        }
        if (!ReadOK(tif->tif_fd, buf, size)) {
            TIFFError(module, "%s: Read error at scanline %d",
                tif->tif_name, tif->tif_row);
            return (-1);
        }
    return (size);
}
```

Having read in the data, we call TIFFStartStrip( ) to set up the pointers used by the decoding routines. If the compression method has a pre-decode routine, we call it here.

```
/*
 * Set state to appear as if a strip has just been read in.
 */
static
TIFFStartStrip(tif, strip)
    register TIFF *tif;
    u_int strip;
{
    TIFFDirectory *td = &tif->tif_dir;

    tif->tif_curstrip = strip;
    tif->tif_row = (strip % td->td_stripsperimage) *
                   td->td_rowsperstrip;
    tif->tif_rawcp = tif->tif_rawdata;
    tif->tif_rawcc = td->td_stripbytecount[strip];
    return (tif->tif_predecode == NULL ||
                        (*tif->tif_predecode)(tif));
}
```

## 18.7 WRITING TIFF DATA

Writing TIFF files is complicated by the fact that the file can and usually does grow during writing. Before it starts writing image data, the application has to set most of the parameters for the file. It has to set the image width and the planar configuration (whether or not components are stored separately). The bits per sample defaults to 1, and compression defaults to none, so the application has to set those as well if they aren't satisfactory. Unless the planar configuration is for separate components, the application does not have to preset the number of rows in the image. This feature can be a convenience in scanner applications that can't tell how long the image is until the scanner reports that it's come to the end of the paper. In the absence of row information, the library computes the number of rows in the image on the basis of the number of rows that the application writes.

The primary entry point for writing a pixel row is TIFFWriteScanline( ). It first checks that the file is OK to write with TIFFWriteCheck( ), then, if necessary, it allocates buffer space with TIFFBufferSetup( ).

```
/* tif_write.c
 * TIFF Library.
 *
 * Scanline-oriented Write Support
 */
#include "tiffioP.h"
#include <stdio.h>
#include <assert.h>

#define STRIPINCR   20      /* expansion factor on strip array */

#if USE_PROTOTYPES
static  TIFFWriteCheck(TIFF *, int, char []);
static  TIFFBufferSetup(TIFF *, char []);
static  TIFFGrowStrips(TIFF *, int, char []);
static  TIFFAppendToStrip(TIFF *, u_int, u_char *, u_int);
#else
/* nonprototype version */
#endif

/*VARARGS3*/
TIFFWriteScanline(tif, buf, row, sample)
    register TIFF *tif;
    u_char *buf;
    u_int row, sample;
{
    static char module[] = "TIFFWriteScanline";
    register TIFFDirectory *td;
    int strip, status, imagegrew = 0;

    if (!TIFFWriteCheck(tif, 0, module))
```

```
        return (-1);
    /*
     * Handle delayed allocation of data buffer.  This
     * permits it to be sized more intelligently (using
     * directory information).
     */
    if ((tif->tif_flags & TIFF_BUFFERSETUP) == 0) {
        if (!TIFFBufferSetup(tif, module))
            return (-1);
        tif->tif_flags |= TIFF_BUFFERSETUP;
    }
    td = &tif->tif_dir;
```

If the image is growing (this row is beyond the current image length) we bump the image length. If this row goes into a different strip from the one we last worked on, we flush any unwritten data with TIFFFlushdata( ).

If this is a new strip beyond the previous end of the image, we update the strips per image tag. Then we figure out which row in the strip we're writing and call the compressor's pre-encode routine to get it ready to compress data for this strip.

```
    /*
     * Extend image length if needed
     * (but only for PlanarConfig=1).
     */
    if (row >= td->td_imagelength) {    /* extend image */
        if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
            TIFFError(tif->tif_name,
        "Cannot change \"ImageLength\" when using separate planes");
            return (-1);
        }
        td->td_imagelength = row+1;
        imagegrew = 1;
    }
    /*
     * Calculate strip and check for crossings.
     */
    if (td->td_planarconfig == PLANARCONFIG_SEPARATE) {
        if (sample >= td->td_samplesperpixel) {
            TIFFError(tif->tif_name,
                "%d: Sample out of range, max %d",
                sample, td->td_samplesperpixel);
            return (-1);
        }
        strip = sample*td->td_stripsperimage
                 + row/td-> td_rowsperstrip;
    } else
        strip = row / td->td_rowsperstrip;
```

```
if (strip != tif->tif_curstrip) {
    /*
     * Changing strips -- flush any data present.
     */
    if (tif->tif_rawcc > 0 && !TIFFFlushData(tif))
        return (-1);
    tif->tif_curstrip = strip;
    /*
     * Watch out for a growing image.  The value of
     * strips/image will initially be 1 (since it
     * can't be deduced until the image length is known).
     */
    if (strip >= td->td_stripsperimage && imagegrew)
        td->td_stripsperimage =
            howmany(td->td_imagelength, td->td_rowsperstrip);
    tif->tif_row =
        (strip % td->td_stripsperimage) *
                                        td->td_rowsperstrip;
    if (tif->tif_preencode && !(*tif->tif_preencode)(tif))
        return (-1);
    tif->tif_flags |= TIFF_POSTENCODE;
                            /* will have to call postencode */
}
```

If we've added a new strip, we have to expand the arrays in the dictionary that track the positions and sizes of the strips, by calling TIFFGrowStrips.

```
/*
 * Check strip array to make sure there's space.
 * We don't support dynamically growing files that
 * have data organized in separate bitplanes because
 * it's too painful.  In that case we require that
 * the imagelength be set properly before the first
 * write (so that the strips array will be fully
 * allocated above).
 */
if (strip >= td->td_nstrips && !TIFFGrowStrips(tif, 1, module))
    return (-1);
```

Now we know that the row we're about to write can exist in the file. We see if we're writing nonsequentially and, if so, call the encoder's seek routine to position to the correct row.

```
/*
 * Ensure the write is either sequential or at the
 * beginning of a strip (or that we can randomly
 * access the data -- i.e., no encoding).
 */
```

```
    if (row != tif->tif_row) {
        if (tif->tif_seek) {
            if (row < tif->tif_row) {
                /*
                 * Moving backward within the same strip:
                 * back up to the start and then decode
                 * forward (below).
                 */
                tif->tif_row = (strip % td->td_stripsperimage) *
                    td->td_rowsperstrip;
                tif->tif_rawcp = tif->tif_rawdata;
            }
            /*
             * Seek forward to the desired row.
             */
            if (!(*tif->tif_seek)(tif, row - tif->tif_row))
                return (-1);
            tif->tif_row = row;
        } else {
            TIFFError(tif->tif_name,
            "Compression algorithm does not support random access");
            return (-1);
        }
    }
```

Finally we call the encode-row routine to compress the image data into the file (actually, the output buffers), bump the row pointer to the next row, and we're done.

```
    status = (*tif->tif_encoderow)(tif, buf, tif->tif_scanlinesize,
                sample);
    tif->tif_row++;
    return (status);
}
```

Before we write a row, several consistency checks are in order. The file has to be opened for writing, and if we're writing rows, the file has to be organized by scan lines, not by tiles.

```
/*
 * Verify file is writable and that the directory
 * information is set up properly.  By doing the latter
 * we also "freeze" the state of the directory so
 * that important information is not changed.
 */
static
TIFFWriteCheck(tif, tiles, module)
    register TIFF *tif;
    int tiles;
```

```
    char module[];
{
    if (tif->tif_mode == O_RDONLY) {
        TIFFError(module, "%s: File not open for writing",
            tif->tif_name);
        return (0);
    }
    if (tiles ^ isTiled(tif)) {
        TIFFError(tif->tif_name, tiles ?
            "Cannot write tiles to a stripped image" :
            "Cannot write scanlines to a tiled image");
        return (0);
    }
```

Until the first image data is written to the file, the user can change any tag in the directory almost without restriction. At the time image writing begins, some tags have to be set so we know how to organize the data being written. Furthermore, many tags can't be changed once any image data is written to the file, since it would require going back and recoding the data already written.

```
/*
 * On the first write verify that all the required information
 * has been set up and initialize any data structures that
 * had to wait until directory information was set.
 * Note that a lot of our work is assumed to remain valid
 * because we disallow any of the important parameters
 * from changing after we start writing (i.e., once
 * TIFF_BEENWRITING is set, TIFFSetField will allow only
 * the image's length to be changed).
 */
if ((tif->tif_flags & TIFF_BEENWRITING) == 0) {
    if (!TIFFFieldSet(tif, FIELD_IMAGEDIMENSIONS)) {
        TIFFError(module,
            "%s: Must set \"ImageWidth\" before writing data",
            tif->tif_name);
        return (0);
    }
    if (!TIFFFieldSet(tif, FIELD_PLANARCONFIG)) {
        TIFFError(module,
        "%s: Must set \"PlanarConfiguration\"
        before writing data",
            tif->tif_name);
        return (0);
    }
    if (tif->tif_dir.td_stripoffset == NULL &&
        !TIFFSetupStrips(tif)) {
        tif->tif_dir.td_nstrips = 0;
        TIFFError(module, "%s: No space for %s arrays",
            tif->tif_name, isTiled(tif) ? "tile" : "strip");
```

```
                return (0);
            }
            tif->tif_flags |= TIFF_BEENWRITING;
        }
        return (1);
}
```

Part of the process of setting up to write image data is to initialize the per-strip data, the stripoffset array that tracks where in the file each strip starts, and the stripbytecount array that tells how long each strip is. We allocate and zero the two arrays, since zero values are interpreted by the rest of the code as "not assigned yet." As we write each strip, we will assign its position and compute its length.

```
static
TIFFSetupStrips(tif)
    TIFF *tif;
{
#define isUnspecified(td, v) \
    (td->v == 0xffffffff || (td)->td_imagelength == 0)
    register TIFFDirectory *td = &tif->tif_dir;

    if (!isTiled(tif))
        td->td_stripsperimage = isUnspecified(td, td_rowsperstrip) ?
            1 : howmany(td->td_imagelength, td->td_rowsperstrip);
    else
        td->td_stripsperimage = isUnspecified(td, td_tilelength) ?
            1 : TIFFNumberOfTiles(tif);
    td->td_nstrips = td->td_stripsperimage;
    if (td->td_planarconfig == PLANARCONFIG_SEPARATE)
        td->td_nstrips *= td->td_samplesperpixel;
    td->td_stripoffset = (u_long *)
        malloc(td->td_nstrips * sizeof (u_long));
    td->td_stripbytecount = (u_long *)
        malloc(td->td_nstrips * sizeof (u_long));
    if (td->td_stripoffset == NULL || td->td_stripbytecount == NULL)
        return (0);
    /*
     * Place data at the end-of-file
     * (by setting offsets to zero).
     */
    bzero((char *)td->td_stripoffset, td->
                        td_nstrips * sizeof (u_long));
    bzero((char *)td->td_stripbytecount, td->td_nstrips *
      sizeof (u_long));
    TIFFSetFieldBit(tif, FIELD_STRIPOFFSETS);
    TIFFSetFieldBit(tif, FIELD_STRIPBYTECOUNTS);
    return (1);
#undef isUnspecified
}
```

We also have to allocate the buffer used for the raw data. The buffer is big enough to hold one scan line and is at least 8K to ensure reasonably fast writes.

```
/*
 * Set up the raw data buffer used for encoding.
 */
static
TIFFBufferSetup(tif, module)
    register TIFF *tif;
    char module[];
{
    int size;

    if (isTiled(tif))
        tif->tif_tilesize = size = TIFFTileSize(tif);
    else
        tif->tif_scanlinesize = size = TIFFScanlineSize(tif);
    /*
     * Make raw data buffer at least 8K
     */
    if (size < 8*1024)
        size = 8*1024;
    tif->tif_rawdata = malloc(size);
    if (tif->tif_rawdata == NULL) {
        TIFFError(module, "%s: No space for output buffer",
            tif->tif_name);
        return (0);
    }
    tif->tif_rawdatasize = size;
    tif->tif_rawcc = 0;
    tif->tif_rawcp = tif->tif_rawdata;
    return (1);
}
```

When the image grows enough that we have to add another strip, TIFFGrowStrips( ) reallocates the two per-strip data structures and zeros the newly added fields.

```
/*
 * Grow the strip data structures by delta strips.
 */
static
TIFFGrowStrips(tif, delta, module)
    TIFF *tif;
    int delta;
    char module[];
{
    TIFFDirectory *td = &tif->tif_dir;
```

```
assert(td->td_planarconfig == PLANARCONFIG_CONTIG);
td->td_stripoffset = (u_long *)realloc(td->td_stripoffset,
    (td->td_nstrips + delta) * sizeof (u_long));
td->td_stripbytecount = (u_long *)realloc(td->td_stripbytecount,
    (td->td_nstrips + delta) * sizeof (u_long));
if (td->td_stripoffset == NULL ||
    td->td_stripbytecount == NULL) {
    td->td_nstrips = 0;
    TIFFError(module, "%s: No space to expand strip arrays",
        tif->tif_name);
    return (0);
}
bzero(td->td_stripoffset+td->td_nstrips, delta*sizeof (u_long));
bzero(td->td_stripbytecount+td->
        td_nstrips, delta*sizeof (u_long));
    td->td_nstrips += delta;
    return (1);
}
```

We start sending data to the file with TIFFFlushData1( ). (It's an internal version of the public routine TIFFFlushData( ), not shown here.) If the file uses the opposite bit order, we reverse the bits in each byte. Then we call TIFFAppendToStrip( ) to write the actual strip data to the file.

```
/*
 * Internal version of TIFFFlushData that can be
 * called by ''encodestrip routines'' w/o concern
 * for infinite recursion.
 */
TIFFFlushData1(tif)
    register TIFF *tif;
{
    if (tif->tif_rawcc > 0) {
        if (tif->tif_dir.td_fillorder != tif->tif_fillorder &&
            (tif->tif_flags & TIFF_NOBITREV) == 0)
            TIFFReverseBits((u_char *)tif->tif_rawdata,
                tif->tif_rawcc);
        if (!TIFFAppendToStrip(tif,
            isTiled(tif) ? tif->tif_curtile : tif->tif_curstrip,
            (u_char *)tif->tif_rawdata, tif->tif_rawcc))
            return (0);
        tif->tif_rawcc = 0;
        tif->tif_rawcp = tif->tif_rawdata;
    }
    return (1);
}
```

Finally we get to the routine that writes the strips of image data to the file. If the strip offset is zero, meaning that the strip has never been written, we put the data at the end of the file and note the location in the strip offset array. If we didn't already know the current offset in the file, we use SeekOK, a wrapper around the lseek( ) system call, to go to the end of the file. (We could do that anyway, but tracking the offset ourselves saves the lseek( ) system calls.)

Then we write the data, update the current file position, and add the amount we just wrote to the current strip size. This routine is called once per strip if the entire strip fits in the 8K buffer, or it may be called several times per strip if the strip is bigger than that.

```
/*
 * Append the data to the specified strip.
 *
 * NB: We don't check that there's space in the
 *     file (i.e., that strips do not overlap).
 */
static
TIFFAppendToStrip(tif, strip, data, cc)
    TIFF *tif;
    u_int strip;
    u_char *data;
    u_int cc;
{
    TIFFDirectory *td = &tif->tif_dir;
    static char module[] = "TIFFAppendToStrip";

    if (td->td_stripoffset[strip] == 0 || tif->tif_curoff == 0) {
        /*
         * No current offset, set the current strip.
         */
        if (td->td_stripoffset[strip] != 0) {
            if (!SeekOK(tif->tif_fd, td->td_stripoffset[strip])) {
                TIFFError(module,
                    "%s: Seek error at scanline %d",
                    tif->tif_name, tif->tif_row);
                return (0);
            }
        } else
            td->td_stripoffset[strip] =
                lseek(tif->tif_fd, 0L, L_XTND);
        tif->tif_curoff = td->td_stripoffset[strip];
    }
    if (!WriteOK(tif->tif_fd, data, cc)) {
        TIFFError(module, "%s: Write error at scanline %d",
            tif->tif_name, tif->tif_row);
        return (0);
    }
```

```
    tif->tif_curoff += cc;
    td->td_stripbytecount[strip] += cc;
    return (1);
}
```

## Other Odds and Ends

A few more routines round out our TIFF library. TIFFGetFileSize( ) gets the size, in bytes, of an open file. On UNIX, DOS, and VMS systems, the fstat( ) system call does the trick.

```
/* tif_compat.c
 * TIFF Library Compatibility Routines.
 */
#include "tiffioP.h"

#if defined(unix) || defined(__unix) || defined(MSDOS) \
                  || defined(VMS)
#include <sys/stat.h>

long
TIFFGetFileSize(fd)
        int fd;
{
        struct stat sb;

        return (fstat(fd, &sb) < 0 ? 0 : sb.st_size);
}
#endif
```

### Bit and Byte Swapping

To handle files written with bit and byte orders that disagree with the local machine, we have conversion routines. TIFFSwabShort( ) and TIFFSwabLong( ) swap the bytes in 2- and 4-byte values.

```
/* tif_swab.c
 * TIFF Library Bit & Byte Swapping Support.
 *
 * XXX We assume short = 16 bits and long = 32 bits XXX
 */

#include "tiffio.h"

#ifndef TIFFSwabShort
TIFFSwabShort(wp)
        unsigned short *wp;
{
```

```
        register unsigned char *cp = (unsigned char *)wp;
        int t;

        t = cp[1]; cp[1] = cp[0]; cp[0] = t;
}
#endif

#ifndef TIFFSwabLong
TIFFSwabLong(lp)
        unsigned long *lp;
{
        register unsigned char *cp = (unsigned char *)lp;
        int t;

        t = cp[3]; cp[3] = cp[0]; cp[0] = t;
        t = cp[2]; cp[2] = cp[1]; cp[1] = t;
}
#endif
```

TIFFSwabArrayOfShort( ) and TIFFSwabArrayOfLong( ) do the same thing for arrays of short or long values.

```
#ifndef TIFFSwabArrayOfShort
TIFFSwabArrayOfShort(wp, n)
        unsigned short *wp;
        register int n;
{
        register unsigned char *cp;
        register int t;

        /* XXX unroll loop some */
        while (n-- > 0) {
                cp = (unsigned char *)wp;
                t = cp[1]; cp[1] = cp[0]; cp[0] = t;
                wp++;
        }
}
#endif

#ifndef TIFFSwabArrayOfLong
TIFFSwabArrayOfLong(lp, n)
        register unsigned long *lp;
        register int n;
{
        register unsigned char *cp;
        register int t;

        /* XXX unroll loop some */
        while (n-- > 0) {
```

```
                cp = (unsigned char *)lp;
                t = cp[3]; cp[3] = cp[0]; cp[0] = t;
                t = cp[2]; cp[2] = cp[1]; cp[1] = t;
                lp++;
        }
}
#endif
```

For formats that use low-to-high bit order rather than the high-to-low bit order that we use, the translation table TIFFBitRevTable[ ] provides the bit-reversed version of all byte values. TIFFNoBitRevTable[ ] is an identity table that makes it easier to write bit-reversing code, so the code can just look up bytes in whichever table a pointer points to. The routine TIFFReverseBits( ) reverses all the bytes in an array by looking them up in the table.

```
/*
 * Bit-reversal tables.  TIFFBitRevTable[<byte>] gives
 * the bit-reversed value of <byte>.  Used in various
 * places in the library when the FillOrder requires
 * bit reversal of byte values (e.g., CCITT Fax 3
 * encoding/decoding).  TIFFNoBitRevTable is provided
 * for algorithms that want an equivalent table that
 * does not reverse bit values.
 */
#if defined(__STDC__) || defined(__EXTENDED__) || USE_CONST
const unsigned char TIFFBitRevTable[256] = {
#else
unsigned char TIFFBitRevTable[256] = {
#endif
    0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
    0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
    0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
    0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
    0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
    0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
    0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
    0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
    0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
    0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
    0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
    0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
    0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
    0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
    0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
    0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
    0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
    0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
    0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
    0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
```

```
    0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
    0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
    0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
    0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
    0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
    0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
    0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
    0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
    0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
    0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
    0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
    0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff
};
#if defined(__STDC__) || defined(__EXTENDED__) || USE_CONST
const unsigned char TIFFNoBitRevTable[256] = {
#else
unsigned char TIFFNoBitRevTable[256] = {
#endif
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
    0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
    0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f,
    0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
    0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f,
    0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f,
    0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
    0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f,
    0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
    0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
    0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
    0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
    0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
    0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
    0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
    0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf,
    0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
    0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf,
    0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6, 0xd7,
    0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
    0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
    0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed, 0xee, 0xef,
    0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
```

```
        0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff,
};

TIFFReverseBits(cp, n)
        register unsigned char *cp;
        register int n;
{
        for (; n > 8; n -= 8) {
                cp[0] = TIFFBitRevTable[cp[0]];
                cp[1] = TIFFBitRevTable[cp[1]];
                cp[2] = TIFFBitRevTable[cp[2]];
                cp[3] = TIFFBitRevTable[cp[3]];
                cp[4] = TIFFBitRevTable[cp[4]];
                cp[5] = TIFFBitRevTable[cp[5]];
                cp[6] = TIFFBitRevTable[cp[6]];
                cp[7] = TIFFBitRevTable[cp[7]];
                cp += 8;
        }
        while (n-- > 0)
                *cp = TIFFBitRevTable[*cp], cp++;
}
```

### Error Handling

The final piece of library code is error and warning handling. The library issues error messages by calling TIFFError( ) and warning messages by calling TIFFWarning( ). In the absence of other arrangements by the application, they print the error message to the standard output. The application can install its own error and warning handlers using TIFFSetErrorHandler( ) and TIFFSetWarningHandler( ), or it can disable warning or error messages by setting the handler to NULL.

In windowed systems where there is no standard error (or it goes to a place not visible to the user) applications should install handlers that pop up the error or warning in a window or otherwise bring it to the user's attention.

```
/* tif_error.c
 * TIFF Library. Error handling
 */
#include <stdio.h>
#include "tiffioP.h"
#include "prototypes.h"

static void
DECLARE3(defaultHandler, char*, module, char*, fmt, va_list, ap)
{
        if (module != NULL)
                fprintf(stderr, "%s: ", module);
        vfprintf(stderr, fmt, ap);
```

```
                fprintf(stderr, ".\n");
        }

        static TIFFErrorHandler _errorHandler = defaultHandler;

        TIFFErrorHandler
        DECLARE1(TIFFSetErrorHandler, TIFFErrorHandler, handler)
        {
                TIFFErrorHandler prev = _errorHandler;
                _errorHandler = handler;
                return (prev);
        }

        void
        #if USE_PROTOTYPES
        TIFFError(char *module, char *fmt, ...)
        #else
        /*VARARGS2*/
        TIFFError(module, fmt, va_alist)
                char *module;
                char *fmt;
                va_dcl
        #endif
        {
                if (_errorHandler) {
                        va_list ap;
                        VA_START(ap, fmt);
                        (*_errorHandler)(module, fmt, ap);
                        va_end(ap);
                }
        }

        /* tif_warning.c
         * TIFF Library. warnings
         */
        #include <stdio.h>
        #include "tiffioP.h"
        #include "prototypes.h"

        static void
        DECLARE3(defaultHandler, char*, module, char*, fmt, va_list, ap)
        {
                if (module != NULL)
                        fprintf(stderr, "%s: ", module);
                fprintf(stderr, "Warning, ");
                vfprintf(stderr, fmt, ap);
                fprintf(stderr, ".\n");
        }
```

```
static TIFFErrorHandler _warningHandler = defaultHandler;

TIFFErrorHandler
DECLARE1(TIFFSetWarningHandler, TIFFErrorHandler, handler)
{
        TIFFErrorHandler prev = _warningHandler;
        _warningHandler = handler;
        return (prev);
}

void
#if USE_PROTOTYPES
TIFFWarning(char *module, char *fmt, ...)
#else
/*VARARGS2*/
TIFFWarning(module, fmt, va_alist)
        char *module;
        char *fmt;
        va_dcl
#endif
{
        if (_warningHandler) {
                va_list ap;
                VA_START(ap, fmt);
                (*_warningHandler)(module, fmt, ap);
                va_end(ap);
        }
}
```

## 18.8 TIFF READING AND APPLICATIONS

Here we sketch out how the library is used in an application. On the companion disk are complete programs to convert between TIFF and PNM formats, which we omit here to save space.

Despite the internal complexity of the library, using it to read TIFF images is quite simple:

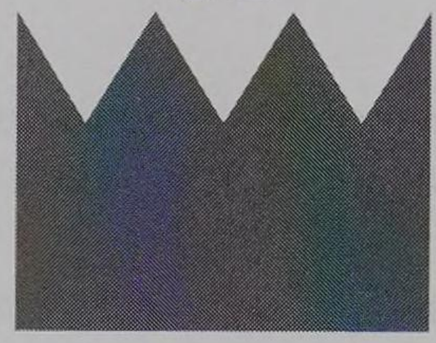
```
#include "tiffio.h"

main(int ac, char **av)
{
        TIFF *tif;
        unsigned long imagelength, imagewidth;
        char *buf;
        long row;

        tif = TIFFOpen(av[1], "r"); /* name on command line */

        /* get the image size so we can allocate buffers */
```

```
        TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &imagelength);
        TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &imagewidth);

        buf = malloc(TIFFScanlineSize(tif));

        /* now read all the data */
        for (row = 0; row < imagelength; row++) {
                TIFFReadScanline(tif, buf, row);
                /* do something with the row data in buf */
        }
}
```

Writing a file is slightly more complex, since we have to set more parameters before we start writing the image data.

```
main(int ac, char **av)
{
        TIFF *tif;
        unsigned long imagelength, imagewidth;
        int bitspersample;          /* bits per component */
        char *buf;
        long row;

        tif = TIFFOpen("myfile.tif", "w");
        TIFFSetField(tif, TIFFTAG_IMAGEWIDTH, imagewidth);
        TIFFSetField(tif, TIFFTAG_IMAGELENGTH, imagelength);
                                                /* optional */
        TIFFSetField(tif, TIFFTAG_COMPRESSION, COMPRESSION_LZW);
    TIFFSetField( tif, TIFFTAG_BITSPERSAMPLE, bitspersample );
    TIFFSetField( tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG );

        /* ... <set other tags> */
    TIFFSetField( tif, TIFFTAG_IMAGEDESCRIPTION, "my TIFF file" );

        /* for color-mapped images */
        {
                unsigned short red[256], green[256], blue[256];

        /* ... compute color map into red[], green[], and blue[] */
                TIFFSetField( tif, TIFFTAG_COLORMAP, red, green,
                blue );
        }

        row = 0;
        while ( /*...more scanlines to write...*/ ) {
                /* put image data in buf */
                TIFFWriteScanline(tif, buf, row);
        }

        TIFFClose(tif);
}
```

CHAPTER 19



# JPEG

JPEG is by far the most sophisticated pixel format we discuss in this book. The companion disk includes a complete and portable JPEG encoder and decoder, but it is so large that it would fill 200 pages of this book. Fortunately, programmers can effectively use the JPEG library without reading the whole thing, so long as they have a general understanding of what JPEG is. We discuss how JPEG coding and decoding work and then look at how an application can use the JPEG library.

## 19.1 WHAT IS JPEG?

JPEG stands for the Joint Photographic Experts Group, a group working under the auspices of the ISO, (the International Organization for Standardization) which is the coordinating body for national and other standards bodies. JPEG is intended to be a standard encoding technique for digitized photographs, a function it serves well.

Unlike all of the other pixel formats we discuss, JPEG is *lossy,* meaning it doesn't necessarily reconstruct the identical pixels in the original full-color image. Even so, the image it does reconstruct looks more like the full-color original than that from "lossless" techniques, such as GIF. Most lossless techniques use a color map with 256 colors or less and approximate the colors in the original image with these 256 colors. This means that GIF doesn't reconstruct an original full-color image either. Instead it constructs a 256-color approximation. In fact the color mapping causes a quality loss much worse than that in a JPEG file of similar size to a GIF file. (Unlike GIF, JPEG can trade off image size against image quality.) A rule of thumb is that JPEG can encode a digitized photograph using an average of 1 bit per pixel.

Some image types are not appropriate for JPEG. Since it was designed to handle photographs, it does not handle well the sharp edges such as those found in

computer-drawn pictures. Nor does it compress large solid color areas particularly well (something at which GIF excels, which is why most GIF images have a solid color border that brings the total image size up to 640 × 480 or 1024 × 768 or another common screen size). For handling photographs, though, it is hard to do better than JPEG.

JPEG is actually designed as a data-stream encoding standard, not a file standard. Hardware JPEG implementations should be fast enough to decode images as they arrive over a communications line, and JPEG has some optional subformats that permit initial display of a low-quality image, with the quality improving as more of the image is received. Software implementations are too slow to keep up with a communication line, so we do not handle the progressive formats.

The JPEG scheme that we discuss here is known as the *baseline* scheme. There is also a lossless mode that has little in common with baseline other than the name and doesn't compress much better than other lossless techniques. (The impetus for the lossless mode appears to be more political than technical.) The lossless scheme is not widely used and we do not handle that either.

### JPEG Formats

Two JPEG file formats have evolved. The simpler is called JFIF, which consists of a JPEG data stream with a few mandatory elements to make sure it can be decoded without having to refer to any external data. Most JPEG or .JPG files found on BBS are actually JFIF files. Unfortunately, some early JPEG compression programs produced incompatible not-quite-JPEG files, but those are now rare.

The other is a new TIFF subformat introduced in the TIFF 6.0 spec, which embeds a JPEG image, possibly divided up into strips or tiles, into a TIFF file. JPEG-in-TIFF has not become popular, probably because it is considerably more complex, but not much more useful, than JFIF and is different enough from other TIFF coding formats that existing TIFF readers and writers are not easily modified to accept it.

## 19.2 STEPS IN JPEG ENCODING

Turning a 24-bit image into a JPEG data stream is a four-step process. The steps are:

- Color coding.
- DCT transformation.
- Quantization.
- Huffman or arithmetic coding.

## Color Coding

The usual first step in compressing a full-color image is color coding. This step is optional but obtains an immediate 50 percent saving of space, so it is almost invariably used.

The human eye is far more sensitive to changes in brightness than it is to changes in color. If we can rearrange RGB data so that the brightness is stored in one component and the color in the other two, it turns out that we can discard most of the color information without affecting the perceived appearance.

The most common color-coding scheme is known as $YC_bC_r$ and is based on that used in color television. The three components are $Y$, which is the luminance of a pixel, and $C_b$ and $C_r$, the chrominance, which can be considered the blueness and redness of the pixel. The actual formulas are:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ C_b &= -0.1687R - 0.3313G + 0.5B \\ C_r &= 0.5R - 0.4187G - 0.0813B \end{aligned}$$

This is a lossless change, since the RGB values can be reconstructed by reversing the arithmetic. Once we have the $YC_bC_r$ components, we *subsample* the chrominance. This means that for each $2 \times 2$ block of pixels, we compute one average $C_b$ and $C_r$ value for the block and just store that. So for each $2 \times 2$ block of pixels, we have 4 $Y$ values, 1 $C_b$ and 1 $C_r$, for a total of 6 components rather than the 12 (4 each of R, G, and B) that we started with, so we have already saved 50 percent of the space required, with only a minimal loss in image quality.

Each of the three components is now treated as a separate image plane and is transformed and compressed separately. The final file has interleaved compressed blocks of the various components, typically four blocks of $Y$ data, one block of $C_b$, one of $C_r$, and so forth.

Color coding doesn't apply to grayscale images, so their pixel values are fed directly into the DCT stage.

## DCT Transformation

The next stage is the Discrete Cosine Transform. This takes advantage of the fact that the eye is much more sensitive to gradual changes in brightness or color than to sudden ones, so it makes sense to store gradual changes more precisely than sudden ones. The DCT lets us do that. It takes an $8 \times 8$ block of input components (either the $Y$ luminance components or a block of already-subsampled $C_b$ or $C_r$ components) and performs a mathematical transform resulting in an $8 \times 8$ array of "spatial frequencies." The (0,0) component in the resulting array is the degree to which the input values don't change at all, which is the same as the average component value. Higher-numbered components reflect higher spatial frequencies, which represent faster changes in brightness or luminance.

For the record, the formula for the DCT follows, with $f(x, y)$ being the component value at $(x, y)$ and $F(u, v)$ being the resulting DCT value in table entry

$(u, v)$. Note that all of the output values depend on all of the intput values; it's not a component-by-component transformation.

$$F(u,v) = \frac{1}{4}C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right]$$

$$\text{where } C(z) = \begin{cases} 1/\sqrt{2} & \text{if } z = 0 \\ 1 & \text{if } z \neq 0 \end{cases}$$

The DCT is still a lossless transformation. The original component values can be recovered by the reverse DCT:

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right]$$

In practice, rounding errors may make the recovered values differ slightly, but that is insignificant compared to quantization effects.

### Quantization

Quantization is the major lossy step in JPEG encoding. Each of the resulting values from the DCT is a 12-bit integer. For any particular JPEG image, there is an 8x8 quantization table that is simply a table of values by which each of the DCT output values is divided. The JPEG decoder has the same table (it's stored with the image) and multiplies the values in the file by the table entry to reconstruct approximately the original DCT data. The larger the divisor, the less exactly the DCT values are reconstructed; but the smaller the quantized value is, the less space it takes in the JPEG file. The (0,0) component has a small divisor around 15, while the highest frequency components have divisors of 100 or more. By choosing an appropriate set of divisors, an application can control the tradeoff between image quality and file size.

### Huffman or Arithmetic Coding

The final step is to take the results of quantization and encode them compactly in a file, a process known as entropy coding. The more common coding option is a variable-length Huffman code with an RLE feature that efficiently represents runs of zeros. (The quantized high-frequency components often end up being zero.)

An alternative coding technique is arithmetic coding, a somewhat more complex but more compact representation. Arithmetic coding is the topic of several patents by IBM and others, and uncertainties about patent licensing have kept it from being widely accepted. The JPEG library uses only Huffman coding.

## 19.3 USING THE JPEG LIBRARY

The JPEG library on the companion disk is designed to be used in two ways—as standalone programs and as library code.

### Standalone Programs

The library code can be built into standalone programs called cjpeg and djpeg, which Code and Decode JPEG files. The input of cjpeg and output of djpeg can be in any of several popular formats including PPM or PGM, GIF, and Targa.

### Library Code

The JPEG library code is designed for extreme flexibility. The basic JPEG encoding and decoding both have many options that the application can specify. It also provides a variety of extra functions which, while not really part of JPEG coding or decoding, are useful for applications that use JPEG, including color quantization of output data (for display on color-mapped screens or writing in color-mapped formats like GIF).

To handle the dozens of possible options, most of which a typical application doesn't care about, the JPEG libraries define a small set of large communication structures.

The major structures are:

**Compress_info_struct** Information needed during compression. Also points to compress methods and external methods.

**Decompress_info_struct** Information needed during decompression. Also points to decompress methods and external methods.

**External_methods_struct** Pointers to common routines used by compress and decompress, primarily error reporting and memory management.

**Compress_methods_struct** Pointers to routines needed for compression.

**Decompress_methods_struct** Pointers to routines needed for decompression.

These structures are allocated in the application, usually as local variables in the routine that invokes JPEG compression or decompression. Most of the fields in the structures have reasonable default values provided by the library, but the application can override them as needed. The library has reasonable default versions for nearly all of the methods needed, which again can be overridden as needed. This is basically object-oriented programming in C; a C++ version of the library would define all of the routines as method routines for the jpegcompress and jpegdecompress classes, that could be subclassed by applications which could then define their own overriding methods in the subclasses.

### JPEG Compression

For JPEG compression, the application has to include both the methods that provide the data to be compressed and some data for the output routine, such as output_file, a FILE pointer to which the output is written (unless the program has provided its own writing methods), the width, height, number of components, and the precision (bits per component) of the input data.

```
/* extracted from jpegdata.h */

struct Compress_info_struct {
        compress_methods_ptr methods;
                /* Points to list of methods to use */

        external_methods_ptr emethods;
                /* Points to list of methods to use */

        IFILEREF input_file;
                /* tells input routines where to read image */
        JFILEREF output_file;
                /* tells output routines where to write JPEG */

        long image_width;          /* input image width */
        long image_height;         /* input image height */
        short input_components;
                    /* # of color components in input image */

        short data_precision;
                        /* bits of precision in image data */

        COLOR_SPACE in_color_space; /* colorspace of input file */

        /* ... many other fields */
};

typedef struct Compress_info_struct * compress_info_ptr;
```

The application must also provide these routines (referred to by the name of the pointer field in the method structure that refers to them):

**input_init** Called to start reading the input. Must set image height, width, number of components, and bits per pixel, and may do any other initialization.

**c_ui_method_selection** Called after input_init, sets the output format methods, typically by calling a utility routine that arranges to write jfif.

**get_input_row** Reads a row of pixels into a library-provided buffer.

**input_term** Cleans up at the end of input.

```
struct Compress_methods_struct {
        /* Hook for user interface to get control after
         * input_init */
        void c_ui_method_selection(compress_info_ptr cinfo);

        /* Input image reading & conversion to standard form */
```

```
                void input_init(compress_info_ptr cinfo);
                void get_input_row(compress_info_ptr cinfo,
                                   JSAMPARRAY pixel_row);
                void input_term(compress_info_ptr cinfo);
                /* ... many other methods */
};
```

Now we show a simplified example of a compression application.

```
/*
 * example.c, sample code from JPEG package
 */

#include "jinclude.h"

/*
 * <setjmp.h> is used for the optional error recovery
 * mechanism shown in the second part of the example.
 */
#include <setjmp.h>
```

The first method is input_init( ), called to set up to read the input. Our simplified example just reads raw RGB pixel data from the input file.

```
METHODDEF void
input_init (compress_info_ptr cinfo)
/* Initialize for input; return image size and component data. */
{
```

This routine must return five pieces of information about the incoming image and must do any setup needed for the get_input_row routine. The image information is returned in fields of the cinfo struct. (If you don't care about modularity, you could initialize these fields in the main JPEG calling routine, and make this routine be a no-op.) We show some example values here.

```
  cinfo->image_width = 640;                 /* width in pixels */
  cinfo->image_height = 480;                 /* height in pixels */
```

JPEG views an image as a rectangular array of pixels, with each pixel having the same number of "component" values (color channels). You must specify how many components there are and the colorspace interpretation of the components. Most applications will use RGB data or grayscale data.

```
  cinfo->input_components = 3;     /* or 1 for grayscale */
  cinfo->in_color_space = CS_RGB;
                        /* or CS_GRAYSCALE for grayscale */
  cinfo->data_precision = 8;
                       /* bits per pixel component value */
}
```

In the current JPEG software, data_precision must be set equal to BITS_IN_JSAMPLE, which is 8 unless you twiddle jconfig.h. Future versions might allow you to say either 8 or 12 if compiled with 12-bit JSAMPLEs, or up to 16 in lossless mode. In any case, it is up to you to scale incoming pixel values to the range 0 .. (1<<data_precision)−1. If your image data format is fixed at a byte per component, then saying "8" is probably the best long-term solution.

The second method actually reads a row of pixels.

This function is called repeatedly and must supply the next row of pixels on each call. The rows *must* be returned in top-to-bottom order if you want your JPEG files to be compatible with everyone else's. (If you cannot readily read your data in that order, you'll need an intermediate array to hold the image. See jrdtarga.c or jrdrle.c for examples of handling bottom-to-top source data using the JPEG code's portable mechanisms.) The data is to be returned into a two-dimensional array of JSAMPLEs, indexed as JSAMPLE pixel_row[component][column] where component runs from 0 to cinfo->input_components−1, and column runs from 0 to cinfo->image_width−1 (column 0 is left edge of image). Note that this is actually an array of pointers to arrays rather than a true two-dimensional array, since C does not support variable-size multidimensional arrays. JSAMPLE is typically typedef'd as "unsigned char".

```
METHODDEF void
get_input_row (compress_info_ptr cinfo, JSAMPARRAY pixel_row)
/* Read next row of pixels into pixel_row[][] */
{
  /*
   * This example shows how you might read RGB data
   * (3 components) from an input file in which the data is
   * stored 3 bytes per pixel in left-to-right, top-to-bottom
   * order.
   */
  register FILE * infile = cinfo->input_file;
  register JSAMPROW ptr0, ptr1, ptr2;
  register long col;

  ptr0 = pixel_row[0];
  ptr1 = pixel_row[1];
  ptr2 = pixel_row[2];
  for (col = 0; col < cinfo->image_width; col++) {
    *ptr0++ = (JSAMPLE) getc(infile); /* red */
    *ptr1++ = (JSAMPLE) getc(infile); /* green */
    *ptr2++ = (JSAMPLE) getc(infile); /* blue */
  }
}
```

The third input method, input_term, cleans up. In many cases, it has no work to do, particularly if the input data comes from an in-memory structure.

```
METHODDEF void
input_term (compress_info_ptr cinfo)
/* Finish up at the end of the input */
{
  /* This termination routine will very often have no work to do,
   * but you must provide it anyway.
   * Note that the JPEG code will call it only during successful
   * exit; if you want it called during error exit, you have to
   * do that yourself. */
}
```

Our final method selects the output format. The simple code below is typical. If the input data is grayscale, it arranges for a grayscale JPEG by calling j_monochrome_default( ). In any event it calls jselwrjfif( ) to arrange to call methods that will write a JFIF file. By providing its own methods, an application can do anything it wants with the JPEG data.

This routine must determine what output JPEG file format is to be written and make any other compression parameter changes that are desirable. This routine gets control after the input file header has been read (i.e., right after input_init has been called). You could combine its functions into input_init, or even into the main control routine, but if you have several different input_init routines, it's a definite win to keep this separate. You *must* supply this routine even if it's a no-op.

```
METHODDEF void
c_ui_method_selection (compress_info_ptr cinfo)
{
  /* If the input is grayscale, generate a monochrome JPEG file. */
  if (cinfo->in_color_space == CS_GRAYSCALE)
    j_monochrome_default(cinfo);
  /* For now, always select JFIF output format. */
  jselwjfif(cinfo);
}
```

The last compression routine is the top-level compressor. It takes one argument, the name of the output file.

```
GLOBAL void
write_JPEG_file (char * filename)
{
```

These three structs contain JPEG parameters and working data. They must survive for the duration of parameter setup and one call to jpeg_compress; typically, making them local data in the calling routine is the best strategy.

```
  struct Compress_info_struct cinfo;
  struct Compress_methods_struct c_methods;
  struct External_methods_struct e_methods;
```

```
/* Initialize the system-dependent method pointers. */
cinfo.methods = &c_methods;         /* links to method structs */
cinfo.emethods = &e_methods;
```

Here we use the default JPEG error handler, which will just print an error message on stderr and call exit( ). See the decompression sample code for an example of more graceful error recovery.

```
jselerror(&e_methods);
                /* select std error/trace message routines */
```

Here we use the standard memory manager provided with the JPEG code. In some cases you might want to replace the memory manager, or at least the system-dependent part of it, with your own code.

```
jselmemmgr(&e_methods);
                    /* select std memory allocation routines */
```

If the compressor requires full-image buffers (for entropy-coding optimization or a noninterleaved JPEG file), it will create temporary files for anything that doesn't fit within the maximum-memory setting. (Note that temporary files are *not* needed if you use the default parameters.) You can change the default maximum-memory setting by changing e_methods.max_memory_to_use after jselmemmgr returns. On some systems you may also need to set up a signal handler to ensure that temporary files are deleted if the program is interrupted. (This is most important if you are on MS-DOS and use the jmemdos.c memory manager back end; it will try to grab extended memory for temp files, and that space will *not* be freed automatically.) See jcmain.c or jdmain.c for an example of signal handler.

```
/* Here, set up pointers to your own routines for input data
 * handling and post-init parameter selection.
 */
c_methods.input_init = input_init;
c_methods.get_input_row = get_input_row;
c_methods.input_term = input_term;
c_methods.c_ui_method_selection = c_ui_method_selection;

/* Set up default JPEG parameters in the cinfo data structure. */
j_c_defaults(&cinfo, 75, FALSE);
```

Note: 75 is the recommended default quality level; you may instead pass a user-specified quality level. Be aware that values below 25 will cause nonbaseline JPEG files to be created (and a warning message to that effect will be emitted on stderr). This won't bother our decoder, but some commercial JPEG implementations may choke on nonbaseline JPEG files. If you want to force baseline

compatibility, pass TRUE instead of FALSE. (If nonbaseline files are fine, but you could do without that warning message, set e_methods.trace_level to −1.)

At this point you can modify the default parameters set by j_c_defaults as needed. For a minimal implementation, you shouldn't need to change anything. See jcmain.c for some examples of what you might change.

Here we select the input and output files. Note that cinfo.input_file is used only if your input reading routines use it; otherwise, you can just make it NULL. *Very important:* use "b" option to fopen( ) if you are on a machine that requires it in order to write binary files.

```
  cinfo.input_file = NULL; /* if no actual input file involved */

  if ((cinfo.output_file = fopen(filename, "wb")) == NULL) {
    fprintf(stderr, "can't open %s\n", filename);
    exit(1);
  }

  /* Here we go! */
  jpeg_compress(&cinfo);

  /* That's it.  Nothin' else to do except close files. */
  /* Here we assume only the output file need be closed. */
  fclose(cinfo.output_file);
}
```

Note: if you want to compress more than one image, we recommend you repeat this whole routine. You *must* repeat the j_c_defaults( )/alter parameters-/jpeg_compress( ) sequence, as some data structures allocated in j_c_defaults are freed upon exit from jpeg_compress.

## JPEG Decompression

JPEG decompression is actually a little more flexible than compression, because the application can ask for a variety of postprocessing, such as block-to-block smoothing (which makes many pictures look better), and color mapping.

The decompression structure contains more fields that are likely to be set by the application. The field quantize_colors can be set to have the library build a color map. The flags two_pass_quantize and use_dithering control how hard it works to produce the color-mapped image, and desired_number_of_colors limits the number of colors in the map, generally 16 for a VGA-type display or 256 for a higher resolution screen or GIF or other color-mapped output file.

The flags do_block_smoothing and do_pixel_smoothing control two optional smoothing steps that can make the reconstructed image look better at the cost of some extra processing time.

```
/* Working data for decompression */

struct Decompress_info_struct {
        decompress_methods_ptr methods;
                /* Points to list of methods to use */

        external_methods_ptr emethods;
                /* Points to list of methods to use */

        JFILEREF input_file;
                /* tells input routines where to read JPEG */
        IFILEREF output_file;
                /* tells output routines where to write image */

        /* these can be set at d_ui_method_selection time: */

        COLOR_SPACE out_color_space; /* colorspace of output */

        boolean quantize_colors;
        /* T if output is a colormapped format */
        /* the following are ignored if not quantize_colors: */
        boolean two_pass_quantize;
                        /* use two-pass color quantization? */
        boolean use_dithering;
                        /* want color dithering? */
        int desired_number_of_colors;
                        /* max number of colors to use */

        boolean do_block_smoothing;
                        /* T = apply cross-block smoothing */
        boolean do_pixel_smoothing;
                        /* T = apply post-upsampling smoothing */

        /*
         * These fields are set by read_file_header or read_scan_header
         */

                long image_width;         /* overall image width */
                long image_height;         /* overall image height */

                short data_precision;
                        /* bits of precision in image data */

                COLOR_SPACE jpeg_color_space; /* colorspace of JPEG file */

                /* for color-mapped output
                int actual_number_of_colors; /* actual number of entries */
                JSAMPARRAY colormap;         /* NULL if not valid */
```

```
        /* ... many other fields */
};

typedef struct Decompress_info_struct * decompress_info_ptr;
```

The required methods are similar to the compression methods.

**d_ui_method_selection** Called after reading the JFIF header, so the parameters of the image are known. Usually sets up the rest of the output methods based on the type of image in the file.

**output_init** Called to set up decoded image output after method selection.

**put_color_map** If the method selection selected color mapping, called before writing any pixel data with the color map.

**put_pixel_rows** Called to write a group of output rows. If the image is grayscale or color-mapped, each row consists of single components. Otherwise each row has pointers to three parallel rows of red, blue, and green—unless, of course, the application provided its own color-manipulation methods, which is unlikely.

**output_term** Cleans up at the end of output.

```
/* Methods used during JPEG decompression. */

struct Decompress_methods_struct {
        void d_ui_method_selection(decompress_info_ptr cinfo);
        /* Hook for user interface to do progress monitoring */
        METHOD(void, progress_monitor, (decompress_info_ptr cinfo,
                                        long loopcounter, long looplimit));

        /* Output image writing */
        void output_init(decompress_info_ptr cinfo);
        void put_color_map(decompress_info_ptr cinfo,
                                        int num_colors,
                                        JSAMPARRAY colormap);
        void put_pixel_rows(decompress_info_ptr cinfo,
                                        int num_rows,
                                        JSAMPIMAGE pixel_data);
        void output_term(decompress_info_ptr cinfo);
        /* ... many other methods */
};
```

We extend the decompression example a little by providing our own error and warning methods to replace the default ones in the library. Applications running in an environment where writing an error message to standard output is inappropriate (such as under a windowing system) will always provide their own error routines.

The external methods structure also includes an application-specified trace level that controls how many trace messages are desired and an array message_parm[ ], which is set with argument values describing the error.

```
struct External_methods_struct {
        void error_exit(const char *msgtext);
        void trace_message(const char *msgtext);

        /*
         * Working data for error/trace facility
         * See macros below for the usage of these variables
         */
        int trace_level; /* level of detail of tracing messages */
/* Use level 0 for important warning messages (nonfatal errors) */
/* Use levels 1, 2, 3 for successively more detailed trace
 * options */
```

For recoverable corrupt-data errors, we emit a warning message and keep going. A surrounding application can check for bad data by seeing if num_warnings is nonzero at the end of processing.

```
        long num_warnings;      /* number of corrupt-data warnings */
        int first_warning_level; /* trace level for first warning */
        int more_warning_level;
                              /* trace level for subsequent warnings */
        int message_parm[8];
                            /* store numeric parms for messages here */
        /* ... many memory management routines */
};
```

Our decompressor starts with the error and trace routines. Each routine is called with a message string containing printf-style percent specifiers that refer to the message_parm[ ] entries. Note that the application has to keep its own private pointer to the external methods structure to gain access to message_parm[ ]. (These example routines actually do the same thing as the library defaults.)

```
/* These static variables are needed by the error routines. */
static jmp_buf setjmp_buffer;         /* for return to caller */
static external_methods_ptr emethods;
                            /* needed for access to message_parm */

METHODDEF void
trace_message (const char *msgtext)
{
  fprintf(stderr, msgtext,
          emethods->message_parm[0], emethods->message_parm[1],
          emethods->message_parm[2], emethods->message_parm[3],
          emethods->message_parm[4], emethods->message_parm[5],
```

```
                emethods->message_parm[6], emethods->message_parm[7]);
  fprintf(stderr, "\n");
                        /* there is no \n in the format string! */
}
```

The error exit routine takes the same message string argument but is not allowed to return to its caller within the library. It usually either exits or uses longjmp( ), as here, to return back to the application. Note the call to the free_all method to release any space in use by the library memory manager.

```
METHODDEF void
error_exit (const char *msgtext)
{
  trace_message(msgtext);                  /* report the error message */
  (*emethods->free_all) ();
                                /* clean up memory allocation & temp files */
  longjmp(setjmp_buffer, 1);    /* return control to outer routine */
}
```

We turn to the decompression methods. The sample output_init routine does nothing; but, as the comment notes, when it is called, all of the descriptive data from the JPEG image is available, such as image size, depth, and number of components.

```
METHODDEF void
output_init (decompress_info_ptr cinfo)
/* This routine should do any setup required */
{
}
```

This routine can initialize for output based on the data passed in cinfo. Useful fields include:

**image_width, image_height** Pretty obvious, I hope.

**data_precision** bits per pixel value; typically 8.

**out_color_space** output colorspace previously requested

**color_out_comps** number of color components in same

**final_out_comps** number of components actually output.

final_out_comps is 1 if quantize_colors is true, else it is equal to color_out-_comps.

If you have requested color quantization, the colormap is *not* yet set. You may wish to defer output initialization until put_color_map is called.

If the output is color-mapped, the library calls put_color_map after output_init and before writing any pixel data.

The colormap data is supplied as a two-dimensional array of JSAMPLEs, indexed as JSAMPLE colormap[component][indexvalue] where component runs from 0 to cinfo->color_out_comps−1, and indexvalue runs from 0 to num_colors −1. Note that this is actually an array of pointers to arrays rather than a true two-dimensional array, since C does not support variable-size multidimensional arrays. JSAMPLE is typically typedef'd as "unsigned char." If you want your code to be as portable as the JPEG code proper, you should always access JSAMPLE values with the GETJSAMPLE( ) macro, which will do the right thing if the machine has only signed chars.

```
METHODDEF void
put_color_map (decompress_info_ptr cinfo, int num_colors,
JSAMPARRAY colormap)
/* Write the color map */
{
  /*
   * You need not provide this routine if you always set
   * cinfo->quantize_colors FALSE; but a safer practice is to
   * provide it and have it just print an error message,
   * like this: */
  fprintf(stderr, "put_color_map called:
                   there's a bug here somewhere!\n");
}
```

Actual pixel data is written by calls to put_pixel_rows. Each call provides one or more rows (usually 8 rows, but code shouldn't count on that) to be written out. If the image is grayscale or color-mapped, there is a single set of components. If it is full-color there are three components; red, green, and blue.

The data is supplied as a three-dimensional array of JSAMPLEs, indexed as JSAMPLE pixel_data[component][row][column] where component runs from 0 to cinfo->final_out_comps−1, row runs from 0 to num_rows−1, and column runs from 0 to cinfo->image_width−1 (column 0 is left edge of image). Note that this is actually an array of pointers to pointers to arrays rather than a true three-dimensional array, since C does not support variable-size multidimensional arrays. JSAMPLE is typically typedef'd as "unsigned char." If you want your code to be as portable as the JPEG code proper, you should always access JSAMPLE values with the GETJSAMPLE( ) macro, which will do the right thing if the machine has only signed chars.

If quantize_colors is true, then there is only one component, and its values are indexes into the previously supplied colormap. Otherwise the values are actual data in your selected output colorspace.

```
METHODDEF void
put_pixel_rows (decompress_info_ptr cinfo, int num_rows,
JSAMPIMAGE pixel_data)
/* Write some rows of output data */
{
```

```
/* This example shows
 * how you might write full-color RGB data (3 components)
 * to an output file in which the data is stored 3 bytes per
 * pixel. */
register FILE * outfile = cinfo->output_file;
register JSAMPROW ptr0, ptr1, ptr2;
register long col;
register int row;

for (row = 0; row < num_rows; row++) {
  ptr0 = pixel_data[0][row];
  ptr1 = pixel_data[1][row];
  ptr2 = pixel_data[2][row];
  for (col = 0; col < cinfo->image_width; col++) {
    putc(GETJSAMPLE(*ptr0), outfile);        /* red */
    ptr0++;
    putc(GETJSAMPLE(*ptr1), outfile);        /* green */
    ptr1++;
    putc(GETJSAMPLE(*ptr2), outfile);        /* blue */
    ptr2++;
  }
 }
}
```

After successfully writing a full image, the library calls output_term to clean up. In this simple example it has no work to do, which is often the case if the decompressed image is being buffered in memory or displayed on a screen.

```
METHODDEF void
output_term (decompress_info_ptr cinfo)
/* Finish up at the end of the output */
{
  /* This termination routine may not need to do anything. */
  /* Note that the JPEG code will call it only during successful */
  /* exit; if you want it called during error exit, */
  /* you have to do that yourself. */
}
```

The final application method is the output method selection routine. This example is fairly typical. If the JPEG file is grayscale, it selects grayscale output, otherwise it accepts the default of RGB.

```
METHODDEF void
d_ui_method_selection (decompress_info_ptr cinfo)
{
  /* if grayscale input, force grayscale output; */
  /* else leave the output colorspace as set by main routine. */
  if (cinfo->jpeg_color_space == CS_GRAYSCALE)
    cinfo->out_color_space = CS_GRAYSCALE;
```

```
  /* select output routines */
  cinfo->methods->output_init = output_init;
  cinfo->methods->put_color_map = put_color_map;
  cinfo->methods->put_pixel_rows = put_pixel_rows;
  cinfo->methods->output_term = output_term;
}
```

Finally we have the main decompression routine, which takes as its argument the name of a file to which to write raw RGB pixel data.

Note that the routine calls setjmp( ) so that it can regain control after an output failure and a longjmp( ) from the error routine above.

```
GLOBAL int
read_JPEG_file (char * filename)
{
```

These three structs contain JPEG parameters and working data. They must survive for the duration of parameter setup and one call to jpeg_decompress; typically, making them local data in the calling routine is the best strategy.

```
  struct Decompress_info_struct cinfo;
  struct Decompress_methods_struct dc_methods;
  struct External_methods_struct e_methods;
```

Here we select the input and output files. In this example we want to open the input file before doing anything else, so that the setjmp( ) error recovery below can assume the file is open. Note that cinfo.output_file is used only if your output handling routines use it; otherwise, you can just make it NULL.

*Very important:* use "b" option to fopen( ) if you are on a machine that requires it in order to read binary files.

```
  if ((cinfo.input_file = fopen(filename, "rb")) == NULL) {
    fprintf(stderr, "can't open %s\n", filename);
    return 0;
  }

  cinfo.output_file = NULL;
                      /* if no actual output file involved */

  /* Initialize the system-dependent method pointers. */
  cinfo.methods = &dc_methods;          /* links to method structs */
  cinfo.emethods = &e_methods;
  /* Here we supply our own error handler; compare to use of
   * standard  error handler in the previous write_JPEG_file
   * example.
   */
  emethods = &e_methods;
```

```
                                /* save struct addr for possible access */
  e_methods.error_exit = error_exit;
                                /* supply error-exit routine */
  e_methods.trace_message = trace_message;
                                /* supply trace-message routine */
  e_methods.trace_level = 0;    /* default = no tracing */
  e_methods.num_warnings = 0;   /* no warnings emitted yet */
  e_methods.first_warning_level = 0;
                               /* display first corrupt-data warning */
  e_methods.more_warning_level = 3;
                                /* but suppress additional ones */

  /* prepare setjmp context for possible exit from error_exit */
  if (setjmp(setjmp_buffer)) {
    /* If we get here, the JPEG code has signaled an error.
     * Memory allocation has already been cleaned up (see
     * free_all  call in error_exit), but we need to close the
     * input file before returning.
     * You might also need to close an output file, etc.
     */
    fclose(cinfo.input_file);
    return 0;
  }
```

Here we use the standard memory manager provided with the JPEG code. In some cases you might want to replace the memory manager, or at least the system-dependent part of it, with your own code.

```
  jselmemmgr(&e_methods);
                        /* select std memory allocation routines */
```

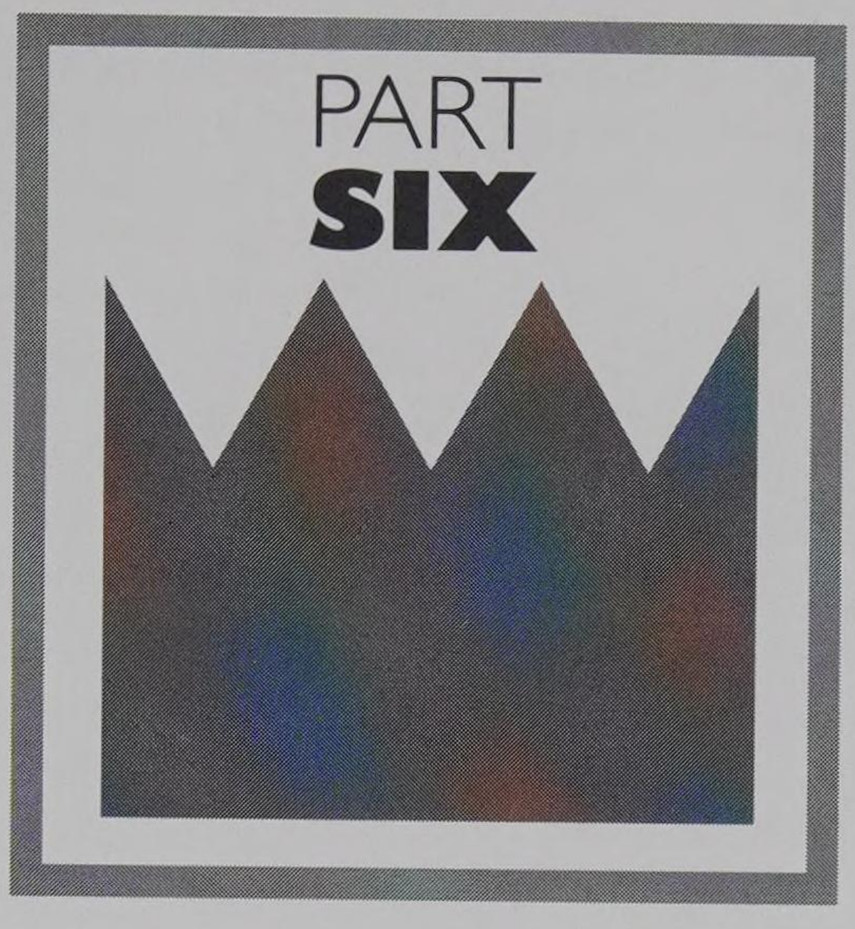
Here, set up the pointer to your own routine for post-header-reading parameter selection. You could also initialize the pointers to the output data handling routines here, if they are not dependent on the image type.

```
  dc_methods.d_ui_method_selection = d_ui_method_selection;

  /* Set up default decompression parameters. */
  j_d_defaults(&cinfo, TRUE);
  /* TRUE indicates that an input buffer should be allocated.
   * In unusual cases you may want to allocate the input buffer
   * yourself; see jddeflts.c for commentary.
   */
```

At this point you can modify the default parameters set by j_d_defaults as needed; for example, you can request color quantization or force grayscale output. See jdmain.c for examples of what you might change.

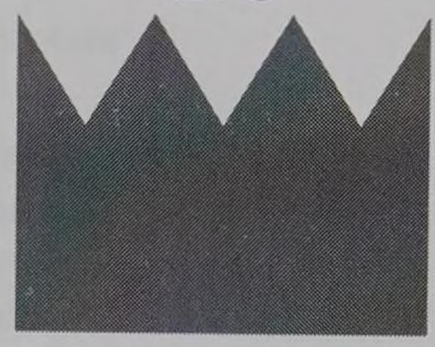
```
    /* Set up to read a JFIF or baseline-JPEG file. */
    /* This is the only JPEG file format currently supported. */
    jselrjfif(&cinfo);

    /* Here we go! */
    jpeg_decompress(&cinfo);

    /* That's it.  Nothin' else to do, except close files. */
    /* Here we assume only the input file need be closed. */
    fclose(cinfo.input_file);

    /* You might want to test e_methods.num_warnings to see if bad
     * data was detected.  In this example, we just blindly forge
     * ahead.
     */
    return 1;                       /* indicate success */
}
```

Note: if you want to decompress more than one image, we recommend you repeat this whole routine. You *must* repeat the j_d_defaults( )/alter parameters/jpeg_decompress( ) sequence, as some data structures allocated in j_d_defaults are freed upon exit from jpeg_decompress.




# Vector Formats

CHAPTER 20



# HP-GL

Hewlett-Packard Graphics Language (HP-GL) was originally introduced in 1976 to control pen plotters. Since then it has become the defacto standard language for control of pen plotters. In 1990 HP introduced HP-GL/2, a much better standardized dialect of HP-GL and implemented it in inkjet and laser printers as well as in pen plotters.

HP-GL is a plain ASCII format and is intended to be extremely easy to write, even for programs written in languages with limited I/O support, such as microcomputer Basic. An HP-GL data stream or file is a series of commands in a uniform format. Each command consists of two uppercase letters, such as PR for Plot using Relative coordinates. Most commands take arguments, which may be numeric or strings. Numeric arguments are separated explicitly by commas or spaces, or implicitly by a + or − starting the next argument. The list of arguments is terminated explicitly with a semicolon or implicitly with the letter that starts the next command. String arguments are terminated explicitly with a marker character, usually ASCII ETX (hex 03).

Pen plotters have traditionally been attached to host systems with serial lines slow enough so that the time it takes to send the plotter commands down the line limits the plotting speed. To speed up plotting, most HP-GL generating programs leave out as many optional characters as possible, so that a set of commands like

```
PD; PR10,20,20,-10,-20,10,-10,-20; PU
```

is most likely sent like this:

```
PDPR10,20,20-10-20,10-10-20PU
```

## 20.1 READING HP-GL FILES

We develop two C++ classes in this section. First we create a low-level class hpglfile that can read commands, numbers, and strings from a C++ input stream.

Using that class, we build some higher-level code that interprets a useful subset of HP-GL commands and tracks the pen position. In Chapter 25 we use this class to build a translator that turns HP-GL files into PPM bitmaps.

## 20.2 LOW-LEVEL HP-GL READER

The C++ class *hpglfile* lets an application read HPGL commands and arguments from a C++ I/O stream. The application creates an hpglfile with a pointer to an open stream. Then class functions read various types of data from the stream. The functions are:

**getcmd** Read a command and return it either as a two-character string or as a code number.

**getnumber** Read a floating point or integer number.

**getstring** Read a string argument.

**getxy** Read a pair of float values.

Each of these routines takes the place to store the value as an argument (or in the case of getxy( ), two arguments) and returns 1 if the value was read successfully or 0 if not.

The header file for the class follows. The private state of the class includes a pointer to the stream from which to read data, a state variable to track where we are in a command, a flag to remember if we've reported on crud in the file, and the delimiter used for string arguments to commands.

```
// hpglfile.h
// HPGL reading class
// John Levine, 4/93

#include <istream.h>

class hpglfile {
    istream *in;          // stream that text comes from
    enum {
        BASE=0,      // haven't read anything
        INARGS,      // reading arguments
        ATEND        // seen semicolon or next cmd
    } state;
    int crudseen;         // only complain once
    int delim;       // string delimiter
public:
    enum commands {
        ERR = 0, CI, PA, PD, PG, PR, PU, SC, SP, UNKCMD
    } lastcmd;
```

```
    hpglfile(istream &fin);                 // constructor

    int getcmd(char *name);                 // get command as a string
    int getcmd(int &code);                  // get command as a code
    int getnumber(float &val);              // get floating arg
    int getnumber(int &val);                // get int arg
    int getstring(char *string, int size);  // get string arg
    int getxy(float &x, float &y);          // get (x,y) arg pair
};
```

The code to implement the class follows. The constructor remembers the stream given as an argument and initializes the other variables.

```
// hpglfile.cpp
// Code for HPGL reading routine
// John Levine, 4/93

#include "hpglfile.h"
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

hpglfile::hpglfile(istream &fin)
{
    in = &fin;
    state = BASE;
    crudseen = 0;
    delim = 03;      // default string delimiter
    lastcmd = ERR;
}
```

Commands are read with getcmd( ). We look first at the version that reads a command as a string. We skip over white space and verify that the next two characters are alphabetic. We make them uppercase if they aren't already and complain if invalid characters are present. We set the state to INARGS, meaning that we're ready to read an argument value.

```
// read an HPGL command from the file, discarding white space

int
hpglfile::getcmd(char *name)
{
    char c;
    int i;

    for(i = 0; i < 2; ) {
        in->get(c);
```

```
    if(!*in)
        return 0;   // EOF or error
    if(i == 0 && isspace(c))
        continue;   // white space
    if(islower(c))
        c = toupper(c);
    else if(!isalpha(c)) {
        // nonalpha, bad stuff in the file
        if(!crudseen) {
            cerr << "Invalid characters in HP-GL stream\n";
            crudseen = 1;
        }
        i = 0;
        continue;
    }
    name[i] = c;
    i++;          // got a good character
  }
  name[2] = 0;    // make a real string
  state = INARGS;
  return 1;       // worked
}
```

The next version of getcmd( ) decodes the command into a code number that is easier for the application to use. The command names are stored in the table cmdlist[ ] in which we look up names with a binary search. The table also contains a flag value describing the type of arguments the function takes, though we don't use that value here.

```
// get HPGL command as a code

int
hpglfile::getcmd(int &code)
{
int low, high;
char name[3];
static struct cmddecode {
    char *name;
    enum cmdtype type;
} cmdlist[] = { // indexed by command type
    { "??", MISCARG},
    { "CI", PAIRARG},
    { "PA", PAIRARG},
    { "PD", PAIRARG},
    { "PG", MISCARG},
    { "PR", PAIRARG},
    { "PU", PAIRARG},
    { "SC", PAIRARG},
    { "SP", PAIRARG},
```

```
};

if(!getcmd(name))
    return 0;

// search for the command in the table
low = 1;
high = UNKCMD - 1;

while(high >= low) {
    int i = (high+low)/2;
    int x = strncmp(name, cmdlist[i].name, 2);

    if(!x) {    // found it
        lastcmd = (enum commands)i;
        lasttype = cmdlist[i].type;
        code = i;
        return 1;
    }
    if(high == low)
        break;
    // not it, try another round
    if(x > 0)
        low = i+1;
    else
        high = i-1;
}
lastcmd = UNKCMD;
return 0;
}
```

The routines getnumber( ) and getstring( ) read argument values from the file. Numeric values are an optional sign followed by a string of digits with an optional decimal point. Arguments are separated explicitly with commas or spaces, or implicitly by a + or − that introduces the next number, and the argument list is terminated with a semicolon or with the first letter of the next command. If we see the end of the argument list, we change the state to ATEND to mark that we can't get any more arguments until we fetch another command.

We read an integer number by the expedient of reading a float and converting it. HP-GL doesn't make a strong distinction between places where floats are allowed and places they aren't, so it seems easier to allow them anywhere and ignore the fractional part if we have to.

The convenience routine getxy merely reads two floats.

```
// get a floating point number
int
hpglfile::getnumber(float &val)
{
```

```
char c;
char buffer[100];
char *bufp = buffer;

if(state != INARGS)
    return 0;                       // no argument here

while(in->get(c)) {
    if(bufp == buffer) {
        if(isspace(c))
            continue;               // ignore white space

        if(c == '+' || c == '-') {  // leading sign OK
            *bufp++ = c;
            continue;
        }
    }
    if(!isdigit(c) && c != '.')
        break;

    *bufp++ = c;                    // OK for number

}
if(c == ';')
    state = ATEND;
else if(isalpha(c)) {
    in->putback(c);                 // next command
    state = ATEND;
}
else if(c == '+' || c == '-') {
    in->putback(c);                 // next argument
}
else if(c != ',' && !isspace(c)) {  // crud in data
    if(!crudseen) {
        cerr << "Invalid characters in HP-GL stream\n";
        crudseen = 1;
    }
}

if(bufp == buffer && state == ATEND)
    return 0;           // wasn't any argument

*bufp = 0;              // return float value
val = atof(buffer);

return 1;
}

// get integer number
```

```
int
hpglfile::getnumber(int &val)
{
    float fval;

    if(getnumber(fval)) {    // worked
        val = (int)fval;
        return 1;
    }
    return 0;
}

// get an x,y pair of arguments
int
hpglfile::getxy(float &x, float &y)
{
    return getnumber(x) && getnumber(y);
}
```

Strings are delimited by a terminator character, usually hex 3. The standard stream library routine get( ) does the trick.

```
int
hpglfile::getstring(char *string, int size)
{
    char c;

    if(state != INARGS)
        return 0;

    if(!in->get(string, size, delim))
        return 0;        // file screwed up

    if(!in->get(c))      // ran off end of file
        return 0;

    if(c != delim) {
        if(!crudseen) {
            cerr << "Unterminated string in HP-GL stream\n";
            crudseen = 1;
        }
    }
    return 1;
}
```

This code is adequate to read any commands in an HP-GL file so long as the enum commands and command table are fleshed out with the rest of the commands.

## 20.3 HIGH-LEVEL HP-GL READER

Using the hpglfile class, we now develop a small program that reads and interprets simple HP-GL files. We handle only a few HP-GL commands, but it turns out that many real HP-GL files use only these basic commands, since the more complex commands tend not to be implemented in exactly the same way on all plotters.

We handle scaling, pen selection, and the four basic pen motion commands; PA, PR, PU, and PD. The SC scaling command takes as its arguments the minimum and maximum $x$ and $y$ coordinates of the plotting area. We translate that to the offset and scale factor for the two axes, which are easier to use for actual scaling. The functions xscale( ) and yscale( ) translate from logical coordinates to physical coordinates, which run from zero to (*x*limit, *y*limit).

```
// readhpgl.cpp
// Read and interpret basic HP-GL
// John Levine, 4/93

#include "hpglfile.h"
#include <fstream.h>
#include <string.h>

float xmin, ymin, xmax, ymax;   // plot limits
float xoffset = 0, xfactor = 1.0;       // scaling info
float yoffset = 0, yfactor = 1.0;

const int xlimit = 500, ylimit = 500;       // size of actual page

inline float xscale(float x) { return (x-xoffset)*xfactor; }
inline float yscale(float y) { return (y-yoffset)*yfactor; }
```

Our program starts by making an ifstream object for the file named on the command line, then making an hpgl object hf.

While we are plotting, there are several important states we have to remember. The obvious ones are the current pen coordinates, curx and cury, and the pen color. Equally important is whether the the pen is down, in which case motion draws lines, or up, in which case it doesn't. The flag dodraw notes whether we're drawing. Also, lines can be drawn either by giving absolute or relative logical coordinates, which we note with the flag dorel.

```
int
main(int ac, char **av)
{
    ifstream fin(av[1]);    // read from input file
    hpglfile hf(fin);
    float arg;
    float xarg, yarg;
    int cmdcode;                        // command we're executing
    int dodraw = 0;         // is the pen down?
```

```
int dorel = 0;          // are we moving relatively?
float curx = 0,         // current pen position
    cury = 0;
float color = 1;        // current pen color
```

We read command by command, switching on the particular command. The SC command takes the minimum and maximum $x$ and $y$ values, from which we compute the $x$ and $y$ offsets and factors.

```
while(hf.getcmd(cmdcode)) {

    switch(cmdcode) {

default:
    cerr << "Mystery code" << cmdcode << endl;
    return 1;

            // set scale
case hpglfile::SC:
    hf.getnumber(xmin); hf.getnumber(xmax);
    hf.getnumber(ymin); hf.getnumber(ymax);

    if(xmin == xmax || ymin == ymax) {
        cerr << "Zero scaling error\n";
        return 1;
    }

    // translate to offset and scale
    // to 0..xlimit in both dimensions
    xoffset = xmin; yoffset = ymin;
    xfactor = xlimit/(xmax-xmin);
    yfactor = ylimit/(ymax-xmin);
    continue;
```

Selecting a pen with SP is easy; we just remember the color. The CI command draws a circle centered at the current point.

```
case hpglfile::SP:
    hf.getnumber(color);
    continue;

case hpglfile::PG:          // Page eject, end of plot
    return 0;

            // draw a circle
case hpglfile::CI:
        xarg = hf.getnumber(xarg);
    cerr << "Circle of radius " << xarg << " at ("
        << curx << "," << cury << ") color " << color << endl;
```

```
            continue;
```

The four commands PA, PD, PR, and PU all can take an arbitrarily long list of $(x,y)$ coordinate pairs. Since they all do more or less the same thing, we set flags appropriately and fall into common code.

```
        case hpglfile::PA:          // Plot Absolute
            dorel = 0;
            break;      // fall into common draw code

        case hpglfile::PD:          // Pen Down
            dodraw = 1;
            break;      // fall into common draw code

        case hpglfile::PR:          // Plot Relative
            dorel = 1;
            break;      // fall into common draw code

        case hpglfile::PU:          // Pen Up
            dodraw = 0;
            break;      // fall into common draw code
            }
```

We read $(x,y)$ pairs until we run out of arguments for this command. If the pen is down, we write out a message about the line we just drew.

```
            // common draw code for PA, PR, PU, PD
            while(hf.getxy(xarg, yarg)) {
                float oldx = curx;
                float oldy = cury;

                if(dorel) {
                    curx += xarg;          // relative coordinates
                    cury += yarg;
                } else {
                    curx = xarg;           // absolute coordinates
                    cury = yarg;
                }
                if(dodraw) {
                    cerr << "draw from (" << xscale(oldx) << ","
                         << yscale(oldy)
                         << ") to (" << xscale(curx) << ","
                         << yscale(cury)
                         << ") color " << color << endl;
                }
            }
        } /* while */
        return 0;
    } /* main */
```

For example, this HP-GL:

```
SC0,1000,0,1000;
PU;SP1;PU;PA200,200PDPR200,200,200,100-200,100,400-500;
```

produces this output:

```
draw from (100,100) to (200,200) color 1
draw from (200,200) to (300,250) color 1
draw from (300,250) to (200,300) color 1
draw from (200,300) to (400,50) color 1
```

In Chapter 25 we interface this code with the PPM rasterization library to produce a program that turns HP-GL into PPM files.

CHAPTER 21

# MS Windows Metafiles

Windows metafiles are a high-level graphics format defined by Microsoft Windows. A metafile is essentially a list of calls to Windows graphics drawing procedures, which has both advantages and disadvantages. The good news is that the format is both compact and flexible. A WMF file is generally much smaller than the equivalent BMP file. The metafile format can encode entire solid color area as a single entry. The bad news is that to draw the image one needs the entire Windows drawing library. If an application is running under Windows anyway, the library is already present and WMF files are easy to display. If not, it's a major challenge to display them at all.

Here we look at the basics of reading and writing WMF files, at least to the point of identifying what calls they are making, along with the code for reading and writing them if you happen to be running under Windows.

## 21.1 READING AND WRITING UNDER MS WINDOWS IN C

Creating a metafile in a Windows C program involves the following steps:

- Create the metafile. Windows returns a "device context" handle used for the rest of the creation process.
- Call the standard graphics routines that a program would use to draw the desired image, but using the metafile's graphics context rather than the context of a screen or printer.
- Close the metafile.

Typical code to create a metafile would be:

```
/* ... */
HANDLE mh;                          /* handle for metafile context */
```

```
HANDLE mh2;                    /* handle for metafile */
HPEN ph;                       /* handle for pen */
extern LOGPEN FAR *mypen;

mh = CreateMetaFile("sample.wmf");

SetWindowOrg(mh, 0, 0);        /* full-screen window */
SetWindowExt(mh, 640,480);

ph = CreatePenIndirect(mypen);
SelectObject(mh, ph);                /* prepare to draw with pen */

Rectangle(mh, 1,1, 639,479);   /* draw frame */
/* ... */

mh2 = CloseMetaFile(mh);             /* close device context,
                                      * create metafile handle */
DeleteMetaFile(mh2);                 /* give back the handle */
```

Despite its name, DeleteMetaFile( ) only deletes the program's handle to the metafile, not the file itself.

Playing back a metafile replays the functions in the metafile but in a context that actually prints or displays something:

```
HANDLE myWindow;     /* application window */
HANDLE mf;           /* the metafile */

myWindow = CreateWindow("my-class", "Sample Application", ...);
...
mf = GetMetaFile("sample.wmf"); /* get handle for metafile */
PlayMetaFile(myWindow, mf);     /* display it */
DeleteMetaFile(mf);             /* give back handle */
```

Windows has some other metafile manipulation routines. Metafiles can exist in memory as well as on the disk, for use as in-memory graphics macros and to pass to the clipboard. The routine EnumMetaFile( ) passes each entry in a metafile to an application program, which can then use PlayMetaFileRecord( ) to play the entries in the metafile one at a time.

## 21.2 UNDER MS WINDOWS IN C++

Microsoft C++ version 7.0 and Visual C++ provide C++ class libraries that make Windows programming somewhat easier than in C.

```
CMetaFileDC mf();              /* create metafile device context */

mf.Create("sample.wmf");       /* create the metafile */
```

```
mh.SetWindowOrg(0, 0);       /* full-screen window */
mh.SetWindowExt(640,480);

extern LOGPEN FAR *mypen;
CPen ph;
ph.CreatePenIndirect(mypen);
mh.SelectObject(ph);              /* prepare to draw with pen */

mh.Rectangle(1,1, 639,479); /* draw frame */
/* ... */

HANDLE mh2 = mh.CloseMetaFile();/* close device context,
                                 * create metafile handle */
DeleteMetaFile(mh2);             /* give back the handle */
```

Playing back the metafile is similarly simpler:

```
/* in a routine where the window is "this" */
...
HANDLE mf = GetMetaFile("sample.wmf");
/* get handle for metafile */
PlayMetaFile(mf);                    /* display it */
DeleteMetaFile(mf);                  /* give back handle */
```

## 21.3 READING METAFILES IN C

For applications that need to read Windows Metafiles, we have the skeleton of a program that reads the records in a metafile.

The header wmf.h defines two ubiquitous Windows datatypes, WORD and DWORD, the two headers found at the front of metafiles, the format of a record in a metafile, and the code numbers for the function types that a metafile record can represent.

```
/*
 *  wmf.h -- Windows metafile structures
 *  John Levine, 4/93
 */

/* two standard windows types */
typedef unsigned short WORD;
typedef unsigned long DWORD;

/* file header */

struct mfheader {
    WORD mtType;
    WORD mtHeaderSize;
    WORD mtVersion;
```

```
    DWORD mtSize;
    WORD mtNoObjects;
    DWORD mtMaxRecord;
    WORD mtNoParameters;
};

struct mfplachdr {
    WORD id1, id2;
    WORD spare1;
    WORD xmin, ymin;
    WORD xmax, ymax;
    WORD scale;
    WORD spare2, spare3;
    WORD xsum;
};

struct mfrec {
    DWORD rdSize;
    WORD rdFunction;
    WORD rdParm[1]; /* rdSize words */
};

/* metafile function codes */

enum {
    /* "typical" functions */
    mfArc        = 0x817,
    mfChord      = 0x830,
    mfEllipse    = 0x418,
    mfExcludeClipRect = 0x415,
    mfFloodFill = 0x419,
    mfIntersectClipRect = 0x416,
    mfLineTo     = 0x213,
    mfMoveTo     = 0x214,
    mfOffsetClipRgn = 0x220,
    mfOffsetViewportOrg = 0x211,
    mfOffsetWindowOrg = 0x20f,
    mfPatBlt     = 0x61d,
    mfPie        = 0x81a,
    mfRealizePalette = 0x035,    /* 3.0 */
    mfRectangle = 0x41b,
    mfResizePalette = 0x139,     /* 3.0 */
    mfRestoreDC = 0x127,
    mfRoundRect = 0x61c,
    mfSaveDC     = 0x01e,
    mfScaleViewportExt = 0x412,
    mfScaleWindowExt = 0x400,
    mfSetBkColor     = 0x201,
    mfSetBkMode = 0x102,
```

```
    mfSetMapMode    = 0x103,
    mfSetMapperFlags = 0x231,
    mfSetPixel  = 0x41f,
    mfSetPolyFillMode = 0x106,
    mfSetROP2   = 0x104,
    mfStretchBltMode = 0x107,
    mfTextAlign = 0x12e,
    mfTextCharExtra = 0x108,
    mfTextColor = 0x209,
    mfTextJustification = 0x20a,
    mfSetViewportExt = 0x20e,
    mfSetViewportOrg = 0x20d,
    mfSetWindowExt  = 0x20c,
    mfSetWindowOrg  = 0x20b,

    /* atypical functions */
    mfAnimatePalette = 0x436,   /* 3.0 */
    mfBitBlt    = 0x922,    /* old */
    mfBitBlt3   = 0x940,    /* 3.0 */
    mfCreateBrushIndirect = 0x2fc,
    mfCreateFontIndirect = 0x2fb,
    mfCreatePalette = 0x0f7,    /* 3.0 */
    mfCreatePatternBrush = 0x1f9,   /* old */
    mfCreatePatternBrush3 = 0x142,  /* 3.0 */
    mfCreatePenIndirect = 0x2fa,
    mfCreateRegion  = 0x6ff,
    mfDeleteObject  = 0x1f0,
    mfDrawText  = 0x62f,
    mfEscape    = 0x626,
    mfExtTextOut    = 0xa32,
    mfPolygon   = 0x324,
    mfPolyPolygon   = 0x538,
    mfPolyline  = 0x325,
    mgSelectClipRegion = 0x12c,
    mgSelectObject  = 0x12d,
    mgSelectPalette = 0x234,
    mgSetDIBitsToDevice = 0xd33,
    mgSetPaletteEntries = 0x037,
    mgStretchBlt    = 0xb23,    /* old */
    mgStretchBlt3   = 0xb41,    /* 3.0 */
    mgStretchDIBits = 0xf43,    /* 3.0 */
    mgTextOut   = 0x521,
};
```

One runs readwmf, passing it the name of a metafile, and it prints out the header and the calls in the metafile.

We start with getword( ) and getdword( ), which fetch WORD and DWORD data from the file independent of byte order. (Windows files are always little-endian since Windows runs primarily on Intel processors.)

```
/*
 *  readwmf.c -- read a WMF file
 *  John Levine, 4/93
 */

#include <stdio.h>
#include "wmf.h"

/* get two bytes */
static WORD
getword(FILE *f)
{
    WORD w;

    w = getc(f);
    w |= getc(f) << 8;
    return w;
} /* getword */

static DWORD
getdword(FILE *f)
{
    DWORD dw;

    dw = (DWORD)getc(f);
    dw |= (DWORD)getc(f) << 8;
    dw |= (DWORD)getc(f) << 16;
    dw |= (DWORD)getc(f) << 24;
    return dw;
}
```

Every metafile starts with one or two headers. First may come the optional placeable metafile header, which can be recognized by the magic number hex cdd7 with which it begins. Following the placeable header, if it is present, or, if not, at the beginning of the file is the regular metafile header that describes the file. It includes the version of Windows that wrote the file (hex 300 for Windows 3), the size of the file in 2-byte words, the number of object handles (index numbers used to identify drawing objects), and the maximum size of any of the records in the file.

```
void
readHeader(FILE *f, struct mfheader *h, struct mfplachdr *mh)
{
    WORD tmp = getword(f);

    if(tmp == 0xcdd7) { /* placeable header */
        WORD csum;
```

```
    mh->id1 = tmp;              csum = tmp;
    mh->id2 = getword(f);           csum ^= mh->id2;
    mh->spare1 = getword(f);        csum ^= mh->spare1;
    mh->xmin = getword(f);          csum ^= mh->xmin;
    mh->ymin = getword(f);          csum ^= mh->ymin;
    mh->xmax = getword(f);          csum ^= mh->xmax;
    mh->ymax = getword(f);          csum ^= mh->ymax;
    mh->scale = getword(f);         csum ^= mh->scale;
    mh->spare2 = getword(f);        csum ^= mh->spare2;
    mh->spare3 = getword(f);        csum ^= mh->spare3;
    mh->xsum = getword(f);          csum ^= mh->xsum;
    if(csum != 0)
        fprintf(stderr, "Warning: bad placeable header\n");

    h->mtType = getword(f);
    } else {
        mh->id1 = 0;     /* note no placeable header */
        h->mtType = tmp;
    }
    h->mtHeaderSize = getword(f);
    h->mtVersion = getword(f);
    h->mtSize = getdword(f);
    h->mtNoObjects = getword(f);
    h->mtMaxRecord = getdword(f);
    h->mtNoParameters = getword(f);
}
```

We read in a metafile object with readObject( ). It returns the object in an allocated buffer or NULL at the end of file.

```
struct mfrec *
readObject(FILE *f)
{
    DWORD size;
    struct mfrec *r;
    int i, nparm;

    size = getdword(f);
    if(feof(f))
        return NULL;     /* end of file */
    r = (struct mfrec *)malloc(sizeof(WORD) * size);
                                        /* size in words */
    r->rdSize = size;
    r->rdFunction = getword(f);
    nparm = size - 3;    /* number of parm words */

    for(i = 0; i < nparm; i++)
        r->rdParm[i] = getword(f);
```

```
    return r;
} /* readObject */
```

We display a metafile object with dumpObject( ). It consists mostly of a big switch that decodes the type of operation. (We can't reasonably use an array of strings, because the function codes are not numbered sequentially.) Then we print out the function arguments as words.

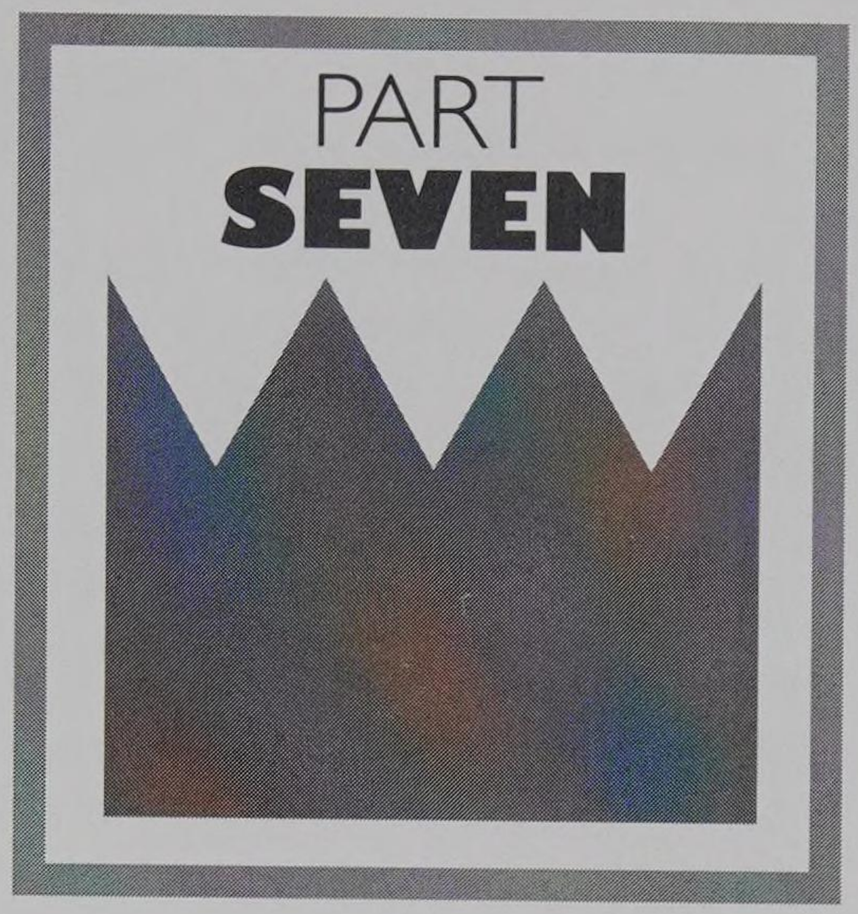
```
dumpObject(struct mfrec *r)
{
    int i, imax;

    switch(r->rdFunction) {
    case mfArc: printf("mfArc"); break;
    case mfChord:   printf("mfChord"); break;
    case mfEllipse: printf("mfEllipse"); break;
    case mfExcludeClipRect: printf("mfExcludeClipRect"); break;
    case mfFloodFill:   printf("mfFloodFill"); break;
    case mfIntersectClipRect:   printf("mfIntersectClipRect");
                                                    break;
    case mfLineTo:  printf("mfLineTo"); break;
    case mfMoveTo:  printf("mfMoveTo"); break;
    case mfOffsetClipRgn:   printf("mfOffsetClipRgn"); break;
    case mfOffsetViewportOrg:   printf("mfOffsetViewportOrg");
                                                    break;
    case mfOffsetWindowOrg: printf("mfOffsetWindowOrg"); break;
    case mfPatBlt:  printf("mfPatBlt"); break;
    case mfPie: printf("mfPie"); break;
    case mfRealizePalette:  printf("mfRealizePalette"); break;
    case mfRectangle:   printf("mfRectangle"); break;
    case mfResizePalette:   printf("mfResizePalette"); break;
    case mfRestoreDC:   printf("mfRestoreDC"); break;
    case mfRoundRect:   printf("mfRoundRect"); break;
    case mfSaveDC:  printf("mfSaveDC"); break;
    case mfScaleViewportExt: printf("mfScaleViewportExt"); break;
    case mfScaleWindowExt:  printf("mfScaleWindowExt"); break;
    case mfSetBkColor:  printf("mfSetBkColor"); break;
    case mfSetBkMode:   printf("mfSetBkMode"); break;
    case mfSetMapMode:  printf("mfSetMapMode"); break;
    case mfSetMapperFlags:  printf("mfSetMapperFlags"); break;
    case mfSetPixel:    printf("mfSetPixel"); break;
    case mfSetPolyFillMode: printf("mfSetPolyFillMode"); break;
    case mfSetROP2: printf("mfSetROP2"); break;
    case mfStretchBltMode:  printf("mfStretchBltMode"); break;
    case mfTextAlign:   printf("mfTextAlign"); break;
    case mfTextCharExtra:   printf("mfTextCharExtra"); break;
    case mfTextColor:   printf("mfTextColor"); break;
    case mfTextJustification:   printf("mfTextJustification");
                                                    break;
```

```
case mfSetViewportExt:  printf("mfSetViewportExt"); break;
case mfSetViewportOrg:  printf("mfSetViewportOrg"); break;
case mfSetWindowExt:    printf("mfSetWindowExt"); break;
case mfSetWindowOrg:    printf("mfSetWindowOrg"); break;

case mfAnimatePalette:  printf("mfAnimatePalette"); break;
case mfBitBlt:  printf("mfBitBlt"); break;
case mfBitBlt3: printf("mfBitBlt3"); break;
case mfCreateBrushIndirect: printf("mfCreateBrushIndirect");
                                                    break;
case mfCreateFontIndirect:  printf("mfCreateFontIndirect");
                                                   break;
case mfCreatePalette:   printf("mfCreatePalette"); break;
case mfCreatePatternBrush:  printf("mfCreatePatternBrush");
                                                   break;
case mfCreatePatternBrush3: printf("mfCreatePatternBrush3");
                                                   break;
case mfCreatePenIndirect:   printf("mfCreatePenIndirect");
                                                   break;
case mfCreateRegion:    printf("mfCreateRegion"); break;
case mfDeleteObject:    printf("mfDeleteObject"); break;
case mfDrawText:    printf("mfDrawText"); break;
case mfEscape:  printf("mfEscape"); break;
case mfExtTextOut:  printf("mfExtTextOut"); break;
case mfPolygon: printf("mfPolygon"); break;
case mfPolyPolygon: printf("mfPolyPolygon"); break;
case mfPolyline:    printf("mfPolyline"); break;
case mgSelectClipRegion:    printf("mgSelectClipRegion");
                                                   break;
case mgSelectObject:    printf("mgSelectObject"); break;
case mgSelectPalette:   printf("mgSelectPalette"); break;
case mgSetDIBitsToDevice:   printf("mgSetDIBitsToDevice");
                                                   break;
case mgSetPaletteEntries:   printf("mgSetPaletteEntries");
                                                   break;
case mgStretchBlt:  printf("mgStretchBlt"); break;
case mgStretchBlt3: printf("mgStretchBlt3"); break;
case mgStretchDIBits:   printf("mgStretchDIBits"); break;
case mgTextOut: printf("mgTextOut"); break;

default:    printf("code%03x", r->rdFunction);
} /* switch */

imax = r->rdSize - 3;

if(!imax)
    printf("( void");
else
    for(i = 0; i < imax; i++)
```

```
            printf("%c %d", (i?',':'('), r->rdParm[i]);
    printf(")\n");
} /* dumpObject */
```

A small main routine reads the header, then reads and prints function records until a function code of zero indicates the end of file.

```
main(int ac, char **av)
{
    FILE *f = fopen(av[1], "r");
    struct mfheader header;
    struct mfplachdr mh;
    struct mfrec *r;

    if(!f) {
        perror(av[1]);
        return 1;
    }

    readHeader(f, &header, &mh);
    printf("Version %x, noobj %d, maxrec %d\n", header.mtVersion,
        header.mtNoObjects, header.mtMaxRecord);
    while(r = readObject(f))  {
        dumpObject(r);
        free(r);
        if(r->rdFunction == 0)
            break;      /* EOF */
    }
} /* main */
```




# Printer Formats

CHAPTER
22

# HP Printer Control Language

The Printer Control Language for the HP Laserjet series (called PCL for short) has become a defacto standard for low- to medium-performance laser printers. PCL consists of plain text, which is printed, along with control sequences that set the print position, select fonts and print orientation, and download bit graphics. Each new generation of printers with PCL has added new features. The original Laserjet was PCL III, and the latest Laserjet 4 models use PCL IV. Fortunately, the changes are almost entirely upward-compatible, so that a file that prints on an original Laserjet will print the same way on the latest model.

The only significant changes to bitmap printing have been the addition of new submodes that handle compressed data using a variety of methods, including the familiar PackBits. We stick with the original scheme both because it is the simplest and because it works on all models of Laserjet.

The native resolution of the Laserjet is 300 DPI. It can print bitmaps at that resolution, or at 150, 100, or 75 DPI by printing each dot as a square 2, 3, or 4 bits on a side. Early Laserjets didn't have enough memory to print a full page at 300 DPI, though more recent ones do. All Laserjets can print a full page at any of the other resolutions. Laserjet bitmaps are single-bit black-and-white images, reflecting the abilities of the hardware. Grayscale can be simulated to some extent by dithering or by treating groups of bits as pseudopixels. For example, if the desired resolution is 150 DPI, a program could print each pixel as a 2x2 square, using five different patterns that made different numbers of bits black, providing five levels of grayscale.

The PCL writing program starts, as usual, with declarations and initialization. There is one flag, to set the resolution used by the printer.

```
/* pbmtolj.c - read a portable bitmap and produce a LaserJet
 * bitmap file */

#include "pbm.h"
```

```
static int dpi = 75;

static void putinit ARGS(( void ));
static void putbit ARGS(( bit b ));
static void putrest ARGS(( void ));
static void putitem ARGS(( void ));

void
main( argc, argv )
    int argc;
    char* argv[];
    {
    FILE* ifp;
    bit* bitrow;
    register bit* bP;
    int argn, rows, cols, format, rucols, padright, row;
    register int nzcol, col;
    char* usage = "[-resolution N] [pbmfile]\n\tresolution =
                  [75|100|150|300] (dpi)";

    pbm_init( &argc, argv );

    argn = 1;

    /* Check for flags. */
    if ( argn < argc && argv[argn][0] == '-'
                                    && argv[argn][1] != '\0' )
    {
    if ( pm_keymatch( argv[argn], "-resolution", 2 ) )
        {
        ++argn;
        if ( argn == argc || sscanf( argv[argn],
                                              "%d", &dpi ) != 1 )
            pm_usage( usage );
        }
    else
        pm_usage( usage );
    ++argn;
    }

    if ( argn != argc )
    {
    ifp = pm_openr( argv[argn] );
    ++argn;
    }
    else
    ifp = stdin;
```

```
if ( argn != argc )
pm_usage( usage );
```

PCL images consist of an escape sequence to set printer parameters and put the printer into raster image mode, followed by a series of rows of pixel data, followed by an end-of-image escape sequence. Since we send the data to the printer a row at a time, we can read the image a row at a time and not buffer the whole image in memory.

We read the image header, allocate a row buffer, and call putinit( ) to send the initialization sequences to the printer.

```
pbm_readpbminit( ifp, &cols, &rows, &format );
bitrow = pbm_allocrow( cols );

putinit( );
```

Each row is sent as an ESC character (hex 1B), followed by *b, the number of bytes of raster data as a decimal number, followed by W and the actual bits, packed eight to a byte with a 1 being black. The rows in an image need not all be the same length, so there is no point in sending anything past the last black pixel in a row.

In each row we scan to find the last black pixel, and round up the length to a multiple of 8 bits. We send the row header followed by the bits, padded to a multiple of 8, using the auxiliary routine putbit( ), below.

At the end of the file we send the end of image sequence with putrest( ) and we're done.

```
for ( row = 0; row < rows; ++row )
{
pbm_readpbmrow( ifp, bitrow, cols, format );

/* Find rightmost black pixel. */
for ( nzcol = cols - 1;
      nzcol >= 0 && bitrow[nzcol] == PBM_WHITE; --nzcol )
      continue;

/* Round up to the nearest multiple of 8. */
rucols = ( nzcol + 8 ) / 8;
rucols = rucols * 8;
padright = rucols - (nzcol + 1);

/* Transfer raster graphics */
printf("\033*b%dW",rucols/8);
    for ( col = 0, bP = bitrow; col <= nzcol; ++col, ++bP )
    putbit( *bP );
for ( col = 0; col < padright; ++col )
    putbit( 0 );
    }
```

```
    pm_close( ifp );

    putrest( );

    exit( 0 );
} /* main */
```

The initialization routine putinit( ) gets the printer ready to print raster graphics. We reset the printer, turn off any top margin to ensure that as much image data fits on the page as possible, set the resolution (75, 100, 150, or 300), and begin the graphics mode. We also set the bit numbers for the bit-packing routine putbit( ).

Bits are packed into bytes by putbit( ). When we've packed 8 bits, we write out the item with putitem( ) and start another byte.

At the end of the image, putrest( ) flushes the last byte if need be and writes the trailing escape sequence.

```
static int item, bitsperitem, bitshift, itemsperline, firstitem;

static void
putinit( )
    {
    /* Printer reset. */
    printf("\033E");  /* not needed in mixed text/graphics */

    /* Ensure top margin is zero */
    printf("\033&l0E"); /* not needed in mixed text/graphics */

    /* Set raster graphics resolution */
    printf("\033*t%dR",dpi);

    /* set upper-left corner position, if not (0,0) */
    /* printf("\033*p%dx%dY", xposition, yposition); */

    /* Start raster graphics, relative addressing */
    printf("\033*r1A");

    itemsperline = 0;
    bitsperitem = 1;
    item = 0;
    bitshift = 7;
    firstitem = 1;
    }

/* add a bit to the output image */
static void
putbit( bit b )
```

```
    {
    if ( b == PBM_BLACK )
    item += 1 << bitshift;
    bitshift--;
    if ( bitsperitem == 8 ) {
    putitem( );
        bitshift = 7;
    }
    bitsperitem++;
    }

/* flush out the current character */
static void
putitem( )
    {
    putchar( item );
    bitsperitem = 0;
    item = 0;
    }

/* flush last image character, leave graphics mode, and reset */
static void
putrest( )
    {
    if ( bitsperitem > 1 )
    putitem( );

    /* end raster graphics */
    printf( "\033*rB" );

    /* Printer reset. */
    printf("\033E"); /* not needed in mixed text/graphics */
    }
```

This routine prints a bitmap image on a full page. Laserjets also support mixing text and graphics. If an image is to be combined with text, omit the resets before and after the image (they start a new page) and the margin-setting command. The top-left corner of the image can be placed at any dot position on the page. Laserjets support positioning in terms of lines and spaces, decipoints (1/720 of an inch) and 300 DPI dots. For graphics, dots are usually the most convenient. The commented-out printf( ) call in putinit( ) will put the image at any desired place.

CHAPTER 23

# PostScript

PostScript is a computer language masquerading as a printer control language. A PostScript printer file is in fact a complete computer program that, as a side effect, prints something. The language contains primitives for drawing lines, arcs, circles, and text with a wide variety of scaling, rotating, and clipping options, and can be thought of as an extremely flexible metafile language. (Display PostScript, a slightly extended version for use on computer screens, has been quite successful as a display drawing language.)

PostScript also contains operators for displaying pixel images, which are what we concentrate on here. The operators are sufficiently general that they can handle grayscale and color images with arbitrary scaling and rotation, and it is PostScript's job to produce the best image that the display or printer hardware is capable of.

Humans tend to find PostScript hard to write. It is a reverse-Polish language—a little like FORTH or an extremely overgrown Hewlett-Packard calculator—so that the arguments to an operator precede the operator itself. The rationale was that most PostScript code is written by other programs (as we do here) rather than by humans, and reverse Polish can be interpreted more efficiently than can other forms.

## 23.1 WRITING POSTSCRIPT

Our converter program turns a PBM, PGM, or PPM image into an Encapsulated PostScript (EPS) file. EPS is a stylized version of PostScript that is easier for non-PostScript applications to handle without having to understand the entire, rather complex PostScript language. EPS files have special comment lines starting with two percent signs that the applications can read to find out basic information about the file such as how big the image is and what text fonts it uses.

The PostScript writer program has a slew of options, including the desired orientation (landscape or portrait), whether to RLE encode the image, the desired height, width, or scaling of the image. An RLE compressed image will be smaller and will take less time to send over a slow serial link to a printer. But since PostScript implementations are interpreted, the more complex decoding program for RLE images runs slower and the total time to print an image is usually about the same, and if the link is fast (e.g., Appletalk or Ethernet) the uncompressed is faster.

Since PostScript can handle either grayscale or color images, this program uses the PNM libraries so that the image being read is kept in its native format.

We start, as usual, by interpreting all of the switches. The values of the switches that set sizes, –width and –height, are multiplied by 72 before being saved, because PostScript's internal unit of distance is the printer's point, 1/72".

```
/* pnmtops.c - read a portable anymap and produce a PostScript file
*/

#include "pnm.h"

#define MARGIN 0.95

static void putinit ARGS(( char* name, int cols, int rows, int
        padright, int bps, float scale, int dpi, int pagewid,
        int pagehgt,
        int format, int turnflag, int turnokflag, int rleflag ));
static void putitem ARGS(( void ));
static void putxelval ARGS(( xelval xv ));
static void putrest ARGS(( void ));
static void rleputbuffer ARGS(( void ));
static void rleputitem ARGS(( void ));
static void rleputxelval ARGS(( xelval xv ));
static void rleflush ARGS(( void ));
static void rleputrest ARGS(( void ));

void
main( argc, argv )
int argc;
char* argv[];
{
    FILE* ifp;
    xel* xelrow;
    register xel* xP;
    int argn,
                turnflag,   /* always print sideways */
                turnokflag, /* OK to print sideways if
                               appropriate */
                rleflag;    /* RLE encode the image */
    int rows, cols, format, bps, padright, row, col;
```

```
    xelval maxval, nmaxval;
    float scale, f;
    int dpi, pagewid, pagehgt;
    char name[100];
    char* cp;
    char* usage = "[-scale <x>] [-turn|-noturn] [-rle|-runlength]
    [-dpi <n>] [-width <n>] [-height <n>] [-rle|-runlength]
    [pnmfile]";

    pnm_init( &argc, argv );

    argn = 1;
    scale = 1.0;
    turnflag = 0;
    turnokflag = 1;
    rleflag = 0;
    /* LaserWriter defaults. */
    dpi = 300;
    pagewid = 612;          /* 8.5 inches in picas */
    pagehgt = 762;          /* 10.58 inches in picas */

    /* Check for flags. */
    while ( argn < argc && argv[argn][0] == '-'
                        && argv[argn][1] != '\0' )
        {
        if ( pm_keymatch( argv[argn], "-scale", 2 ) )
            {
            ++argn;
            if ( argn == argc || sscanf( argv[argn], "%f",
                &scale ) != 1 )
                pm_usage( usage );
            }
        else if ( pm_keymatch( argv[argn], "-turn", 2 ) )
            turnflag = 1;                       /* print sideways */
        else if ( pm_keymatch( argv[argn], "-noturn", 2 ) )
            turnokflag = 0;             /* never print sideways */
        else if ( pm_keymatch( argv[argn], "-rle", 2 ) ||
                  pm_keymatch( argv[argn], "-runlength", 2 ) )
            rleflag = 1;
        else if ( pm_keymatch( argv[argn], "-dpi", 2 ) )
            {
            ++argn;       /* set DPI of output device if not 300 */
            if ( argn == argc || sscanf( argv[argn], "%d", &dpi )
               != 1 )
                pm_usage( usage );
            }
        else if ( pm_keymatch( argv[argn], "-width", 2 ) )
            {
            ++argn;
```

```
        if ( argn == argc || sscanf( argv[argn], "%f", &f )
            != 1 )
            pm_usage( usage );
        pagewid = f * 72.0;
        }
    else if ( pm_keymatch( argv[argn], "-height", 2 ) )
        {
        ++argn;
        if ( argn == argc || sscanf( argv[argn], "%f", &f )
            != 1 )
            pm_usage( usage );
        pagehgt = f * 72.0;
        }
    else
        pm_usage( usage );
        ++argn;
        }
```

We open the file and save its name, minus any extension, so we can put the name in the EPS header.

```
if ( argn < argc )
    {
    ifp = pm_openr( argv[argn] );
    strcpy( name, argv[argn] );
    if ( strcmp( name, "-" ) == 0 )
        strcpy( name, "noname" );

    if ( ( cp = index( name, '.' ) ) != 0 )
        *cp = '\0';
    ++argn;
    }
else
    {
    ifp = stdin;
    strcpy( name, "noname" );
    }

if ( argn != argc )   /* extra arguments, complain */
    pm_usage( usage );
```

We read the header of the input file to figure out the size and depth of the image. PostScript image data can have 1, 2, 4, or 8 bits per pixel or pixel component, so in bps we put the smallest size large enough to handle all of the values in the image.

Each image row has to start on a byte boundary, so we compute the padding needed at the end of each row.

```
pnm_readpnminit( ifp, &cols, &rows, &maxval, &format );
```

```
    xelrow = pnm_allocrow( cols );

    /* Figure out bps. */
    bps = pm_maxvaltobits( (int) maxval );
    if ( bps > 2 && bps < 4 )
        bps = 4;
    else if ( bps > 4 && bps < 8 )
        bps = 8;
    else if ( bps > 8 )
        pm_error( "maxval of %d is too large for PostScript",
        maxval );
    nmaxval = pm_bitstomaxval( bps );

    /* Compute padding to round cols * bps up to the nearest
     * multiple of 8. */
    padright = ( ( ( cols * bps + 7 ) / 8 ) * 8 - cols * bps )
    / bps;
```

Now we write out the image. We call putinit( ), below, to write the header that precedes image data. The details of writing the image data differ depending on whether the image is color or grayscale; the latter includes black-and-white PBM images. Grayscale image rows are written as a series of packed pixels. Color image rows are written as a row of red components, then a row of green components, then a row of blue components. PostScript lets you order the data any way you want by suitable PostScript programming, but a program that read pixels in a more natural RGB format would run much more slowly.

For each row, we read the row, rescale the data if the number of bits used for the PostScript image is greater than the natural size of the data, and write out the row in one pass for grayscale, three passes for color. We call putxelval( ) to write a pixel in an uncompressed image and rleputxelval( ) to write one in an RLE image. In the latter case, rleflush( ) forces out any buffered data at the end of a row.

```
    putinit(
        name, cols, rows, padright, bps, scale, dpi, pagewid,
        pagehgt, format,
        turnflag, turnokflag, rleflag );
    for ( row = 0; row < rows; ++row )
        {
        pnm_readpnmrow( ifp, xelrow, cols, maxval, format );
        switch ( PNM_FORMAT_TYPE( format ) )
            {
            case PPM_TYPE:
            /* Color. */
            for ( col = 0, xP = xelrow; col < cols; ++col, ++xP )
                if ( maxval != nmaxval )
                    PPM_DEPTH( *xP, *xP, maxval, nmaxval );
            /* First red. */
```

```
for ( col = 0, xP = xelrow; col < cols; ++col, ++xP )
            if ( rleflag )
                rleputxelval( PPM_GETR( *xP ) );
            else
                putxelval( PPM_GETR( *xP ) );
for ( col = 0; col < padright; ++col )
            if ( rleflag )
                rleputxelval( 0 );
            else
                putxelval( 0 );
if ( rleflag )
            rleflush();

/* Then green. */
for ( col = 0, xP = xelrow; col < cols; ++col, ++xP )
            if ( rleflag )
                rleputxelval( PPM_GETG( *xP ) );
            else
                putxelval( PPM_GETG( *xP ) );
for ( col = 0; col < padright; ++col )
            if ( rleflag )
                rleputxelval( 0 );
            else
                putxelval( 0 );
if ( rleflag )
            rleflush();

/* And blue. */
for ( col = 0, xP = xelrow; col < cols; ++col, ++xP )
            if ( rleflag )
                rleputxelval( PPM_GETB( *xP ) );
            else
                putxelval( PPM_GETB( *xP ) );
for ( col = 0; col < padright; ++col )
            if ( rleflag )
                rleputxelval( 0 );
            else
                putxelval( 0 );
if ( rleflag )
            rleflush();
break;

default:
/* Grayscale. */
for ( col = 0, xP = xelrow; col < cols; ++col, ++xP )
    {
    if ( maxval != nmaxval )
        PNM_ASSIGN1( *xP, (int) PNM_GET1(*xP) *
```

```
                nmaxval / maxval );
            if ( rleflag )
                rleputxelval( PNM_GET1( *xP ) );
            else
                putxelval( PNM_GET1( *xP ) );
            }

        for ( col = 0; col < padright; ++col )
                if ( rleflag )
                    rleputxelval( 0 );
                else
                    putxelval( 0 );

        if ( rleflag )
                rleflush();
                break;
                }
        } /* for rows */
```

Having written the data, we call putrest( ) or rleputrest( ) to write the trailer.

```
    pm_close( ifp );

    if ( rleflag )
            rleputrest();
    else
            putrest();

    exit( 0 );
}
```

The routine putinit( ), which takes an excessive number of arguments, writes out a PostScript program to print the image.

```
static int bitspersample, item, bitsperitem, bitshift,
       itemsperline,
       items;
static int rleitem, rlebitsperitem, rlebitshift;
static int repeat, itembuf[128], count, repeatitem, repeatcount;

static void
putinit( char* name, int cols, int rows, int padright, int bps,
         float scale,
         int dpi, int pagewid, int pagehgt, int format,
         int turnflag,
         int turnokflag, int rleflag )
{
    int icols, irows, devpix;
    float pixfac, scols, srows, llx, lly;
```

If the user said to print sideways, or if the image is wider than it is high, and the user didn't say not to do so, we set turnflag to note that we need to have the PostScript program turn the image sideways.

```
/* Turn? */
icols = cols;
irows = rows;
if ( turnflag || ( turnokflag && cols > rows ) )
    {
        turnflag = 1;
        cols = irows; /* switch horiz and vertical sizes */
        rows = icols;
    }
```

Several factors control the scaling. The first is that the image has to fit on the page, 8.5 by 10.5 inches unless overriden by command-line flags. The other is that we want the number of printer dots per pixel to be an integer to avoid unattractive artifacts in the printed image.

We solve the second problem first, by first computing devpix, the number of printer dots per point, rounded to an integer, then by inverting that calculation to come up with pixfac, a number close to 1.0, which will produce an integer number of dots when multiplied by an integral height or width. Then we check the height and width of the image and rescale if it doesn't fit.

```
/* Figure out size. */
devpix = dpi / 72.0 + 0.5; /* device pixels per unit, approx. */
pixfac = 72.0 / dpi * devpix;        /* 1, approx. */

scols = scale * cols * pixfac;        /* scaled columns */
srows = scale * rows * pixfac;        /* scaled rows */
if ( scols > pagewid * MARGIN || srows > pagehgt * MARGIN )
    {
    if ( scols > pagewid * MARGIN )
        {
        scale *= pagewid / scols * MARGIN;
        scols = scale * cols * pixfac;
        srows = scale * rows * pixfac;
        }
    if ( srows > pagehgt * MARGIN )
        {
        scale *= pagehgt / srows * MARGIN;
        scols = scale * cols * pixfac;
        srows = scale * rows * pixfac;
        }
    pm_message(
        "warning, image too large for page, rescaling to %g",
         scale );
    }
```

We compute llx and lly, the *x* and *y* margins that will center the image on the page, and write out some EPS headers with the name of the picture and the *bounding box*, the limits of the printed area on the page. (PostScript interpreters don't care about the bounding box, but page makeup and word processing programs find it very useful when embedding EPS images in larger documents.)

```
llx = ( pagewid - scols ) / 2;
lly = ( pagehgt - srows ) / 2;

printf( "%%!PS-Adobe-2.0 EPSF-2.0\n" );
printf( "%%%%Creator: pnmtops\n" );
printf( "%%%%Title: %s.ps\n", name );
printf( "%%%%Pages: 1\n" );
printf(
    "%%%%BoundingBox: %d %d %d %d\n",
    (int) llx, (int) lly,
    (int) ( llx + scols + 0.5 ), (int) ( lly + srows + 0.5 ) );
printf( "%%%%EndComments\n" );
```

The PostScript image printing operators get their pixel data by calling a user-provided read routine, which we call readstring. If we are RLE compressing the image, we write out an RLE decoding program that reads hex RLE bytes and expands them. If not, we write out a much smaller plain version that reads straight hex data.

```
if ( rleflag )
    {
    printf( "/rlestr1 1 string def\n" );
    printf( "/readrlestring {\n" );      /* s -- nr */
    printf( "  /rlestr exch def\n" );    /* - */
    printf( "  currentfile rlestr1 readhexstring
               pop\n" );
                                                /* s1 */
    printf( "  0 get\n" );                      /* c */
    printf( "  dup 127 le {\n" );               /* c */
    printf( "    currentfile rlestr 0\n" );          /* c f s 0 */
    printf( "    4 3 roll\n" );                 /* f s 0 c */
    printf( "    1 add  getinterval\n" );          /* f s */
    printf( "    readhexstring pop\n" );          /* s */
    printf( "    length\n" );                   /* nr */
    printf( "  } {\n" );                        /* c */
    printf( "    256 exch sub dup\n" );         /* n n */
    printf( "    currentfile rlestr1 readhexstring
               pop\n" );
                                               /* n n s1 */
    printf( "    0 get\n" );                   /* n n c */
    printf( "    exch 0 exch 1 exch 1 sub {\n" );
                                               /* n c 0 1 n-1*/
```

```
        printf( "      rlestr exch 2 index put\n" );
        printf( "    } for\n" );              /* n c */
        printf( "    pop\n" );                    /* nr */
        printf( "  } ifelse\n" );              /* nr */
        printf( "} bind def\n" );
        printf( "/readstring {\n" );              /* s -- s */
        printf( "  dup length 0 {\n" );             /* s l 0 */
        printf( "    3 copy exch\n" );              /* s l n s n l*/
        printf( "    1 index sub\n" );              /* s l n s n r*/
        printf( "    getinterval\n" );              /* s l n ss */
        printf( "    readrlestring\n" ); /* s l n nr */
        printf( "    add\n" );                      /* s l n */
        printf( "    2 copy le { exit } if\n" ); /* s l n */
        printf( "  } loop\n" );                             /* s l l */
        printf( "  pop pop\n" );                      /* s */
        printf( "} bind def\n" );
        }
    else
        {
        printf( "/readstring {\n" );                  /* s -- s */
        printf( "  currentfile exch readhexstring pop\n" );
        printf( "} bind def\n" );
        }
```

PostScript requires that the buffers into which the image data are read have been predeclared as PostScript strings. For color data, we declare 3 strings rpicstr, gpicstr, and bpicstr, each long enough to hold a row of component data. For grayscale, a single buffer called picstr suffices.

This ends the prolog part of the file, the part which, if the image is embedded in a larger document, can be moved to the front of the larger document.

```
if ( PNM_FORMAT_TYPE( format ) == PPM_TYPE )
    {
    printf( "/rpicstr %d string def\n", ( icols + padright )
                                        * bps / 8 );
    printf( "/gpicstr %d string def\n", ( icols + padright )
                                        * bps / 8 );
    printf( "/bpicstr %d string def\n", ( icols + padright )
                                        * bps / 8 );
    }
else
    printf( "/picstr %d string def\n", ( icols + padright )
                                        * bps / 8 );
printf( "%%%%EndProlog\n" );
printf( "%%%%Page: 1 1\n" );
```

Now we write out the actual printing program, which is quite small. A gsave operator saves the current graphical state in the printer, which includes the current

scale factor, rotation angle, etc. We write out a translate operator to shift the coordinate system so that the image is in the center of the page, then a scale operator to make the image the appropriate size. If turnflag is set, we output a rotate operator to turn the image 90 degrees. The translate operators before and after the rotate make the center of rotation slightly outside the image, which looks better.

Then we output the parameters for the image or colorimage operator, and the operator itself. This ends the header; the calls to our readstring routine will read hex pixel data, starting on the next line of the file.

```
printf( "gsave\n" );
printf( "%g %g translate\n", llx, lly );
printf( "%g %g scale\n", scols, srows );

if ( turnflag )
        printf( "0.5 0.5 translate  90 rotate
        -0.5 -0.5 translate\n" );

printf( "%d %d %d\n", icols, irows, bps );
printf( "[ %d 0 0 -%d 0 %d ]\n", icols, irows, irows );
if ( PNM_FORMAT_TYPE( format ) == PPM_TYPE )
    {
        printf( "{ rpicstr readstring }\n" );
        printf( "{ gpicstr readstring }\n" );
        printf( "{ bpicstr readstring }\n" );
        printf( "true 3\n" );
        printf( "colorimage\n" );
        pm_message( "writing color PostScript..." );
    }
else
    {
        printf( "{ picstr readstring }\n" );
        printf( "image\n" );
    }
```

The initialization routine ends by setting up some static variables used by the pixel output routines. PostScript reads pixel data as hex bytes, so we have to be prepared to pack the data into bytes, then output the bytes in hex.

```
bitspersample = bps;
itemsperline = items = 0;
if ( rleflag )
    {
        rleitem = 0;
        rlebitsperitem = 0;
        rlebitshift = 8 - bitspersample;
        repeat = 1;
    count = 0;
```

```
        }
    else
        {
                item = 0;
                bitsperitem = 0;
                bitshift = 8 - bitspersample;
        }
} /* putinit */
```

We write a pixel or component with putxelval( ). It shifts the pixel into the current output byte. If the byte is full, it calls putitem( ) to write the byte out.

The basic routine to output a byte of data is putitem( ). It writes the byte out as 2 hex digits, starting a new text line every 30 bytes (60 characters.)

```
static void
putxelval( xelval xv )
{
    if ( bitsperitem == 8 )
                putitem();
    item += xv << bitshift;
    bitsperitem += bitspersample;
    bitshift -= bitspersample;
}

static void
putitem()
{
    char* hexits = "0123456789abcdef";

    if ( itemsperline == 30 )
                        /* 60 chars per line, make it look nice */
        {
                putchar( '\n' );
                itemsperline = 0;
        }
    putchar( hexits[item >> 4] );
    putchar( hexits[item & 15] );
    ++itemsperline;
    ++items;
    item = 0;
    bitsperitem = 0;
    bitshift = 8 - bitspersample;
}
```

At the end of an uncompressed image, we call putrest( ). First, it flushes any partial byte. Then it writes out a call to the PostScript routine grestore, which restores the scaling, translation, and rotation in effect at the gsave call in the

prolog, and a call to showpage, which actually prints the image. An EPS trailer line ends the image.

```
static void
putrest()
{
    if ( bitsperitem > 0 )
                putitem();
    printf( "\n" );
    printf( "grestore\n" );
    printf( "showpage\n" );
    printf( "%%%%Trailer\n" );
}
```

RLE output is a little more complex. The RLE compression is in fact the familiar PackBits.

We call rleputxelval( ) to output a pixel. It packs the pixel into a byte and calls rleputitem( ) if the byte is full.

```
static void
rleputxelval( xelval xv )
{
    if ( rlebitsperitem == 8 )
                rleputitem();
    rleitem += xv << rlebitshift;
    rlebitsperitem += bitspersample;
    rlebitshift -= bitspersample;
}
```

The heart of RLE compression is rleputitem( ). It tracks the number of repeated bytes seen since the last output, in repeat, or the number of nonrepeating bytes buffered, with the count in count and the buffer in itembuf[ ].

The code is basically the same as the PackBits code used for encoding MacPaint or PackBits TIFF images. See Chapters 11 or 18.

```
static void
rleputitem()
{
    int i;

    if ( count == 128 )                    /* max run length */
                rleputbuffer();

    if ( repeat && count == 0 )
        { /* Still initializing a repeat buf. */
                itembuf[count] = repeatitem = rleitem;
                ++count;
        }
```

```
        else if ( repeat )
            { /* Repeating - watch for end of run. */
            if ( rleitem == repeatitem )
                { /* Run continues. */
                itembuf[count] = rleitem;
                ++count;
                }
            else
                { /* Run ended - is it long enough to dump? */
                if ( count > 2 )
                    { /* Yes, dump a repeat-mode buffer and
                       * start a new one. */
                    rleputbuffer();
                    itembuf[count] = repeatitem = rleitem;
                    ++count;
                    }
                else
                    { /* Not long enough - convert to nonrepeat mode. */
                    repeat = 0;
                    itembuf[count] = repeatitem = rleitem;
                    ++count;
                    repeatcount = 1;
                    }
                }
            }
        else
            { /* Not repeating - watch for a run worth
               * repeating. */
            if ( rleitem == repeatitem )
                { /* Possible run continues. */
                ++repeatcount;
                if ( repeatcount > 3 )
                    { /* Long enough - dump nonrepeat part and start
                       * repeat. */
                    count = count - ( repeatcount - 1 );
                    rleputbuffer();
                    count = repeatcount;
                    for ( i = 0; i < count; ++i )
                        itembuf[i] = rleitem;
                    }
                else
                    { /* Not long enough yet - continue as nonrepeat
                       * buf. */
                    itembuf[count] = rleitem;
                    ++count;
                    }
                }
            else
                { /* Broken run. */
```

```
                itembuf[count] = repeatitem = rleitem;
                ++count;
                repeatcount = 1;
                }
            }

        rleitem = 0;
        rlebitsperitem = 0;
        rlebitshift = 8 - bitspersample;
} /* rleputitem */
```

When we have a complete repeated or nonrepeated run, rleputbuffer( ) writes it out, with a negative count and a single data byte for repeated data or a positive count and the actual bytes for nonrepeated data.

At the end of each row, rleflush( ) forces out any pending run. At the end of the file, rleputrest( ) calls rleflush( ) and then does the same thing as putrest( ).

```
static void
rleputbuffer()
{
    int i;

    if ( repeat )
        {
                item = 256 - count;
                putitem();
                item = repeatitem;
                putitem();
        }
    else
        {
                item = count - 1;
                putitem();
                for ( i = 0; i < count; ++i )
                    {
                        item = itembuf[i];
                        putitem();
                    }
            }
        repeat = 1;
        count = 0;
}

static void
rleflush()
{
    if ( rlebitsperitem > 0 )
        rleputitem();
    if ( count > 0 )
```

```
            rleputbuffer();
}

static void
rleputrest()
{
    rleflush();
    printf( "\n" );
    printf( "grestore\n" );
    printf( "showpage\n" );
    printf( "%%%%Trailer\n" );
}
```

## 23.2 READING POSTSCRIPT

Reading and interpreting PostScript files is a very difficult task—one needs an entire PostScript implementation to do so completely. On a system running Display PostScript, it is possible to tell the interpreter to send the output bits to a bitmap area separate from the visible screen, run the PostScript file of interest, and read back the bitmap to recover the image.

Both commercial and noncommercial PostScript interpreters are available. One popular one (because it's free) is Ghostscript, available in C source form for a handling fee from the Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA.

### Reading EPS Preview Data

Fortunately, Encapsulated PostScript comes to our rescue. An EPS Interchange, or EPSI file, is an EPS file that contains a preview bitmap as an EPS header. The preview is identified by a header line of the form:

```
%%BeginPreview:  width height bps lines
```

where width and height are the size of the image in pixels; bps is the bits per pixel, which must be 1, 2, 4, or 8; and lines is the number of text lines of hex data that follow. The image is hex data in the same format as in the uncompressed grayscale images we just saw how to write. Each line of hex data starts with percent signs to make it a comment in the main PostScript program. For example, a $3 \times 3$ box with a black border and white center would look like this:

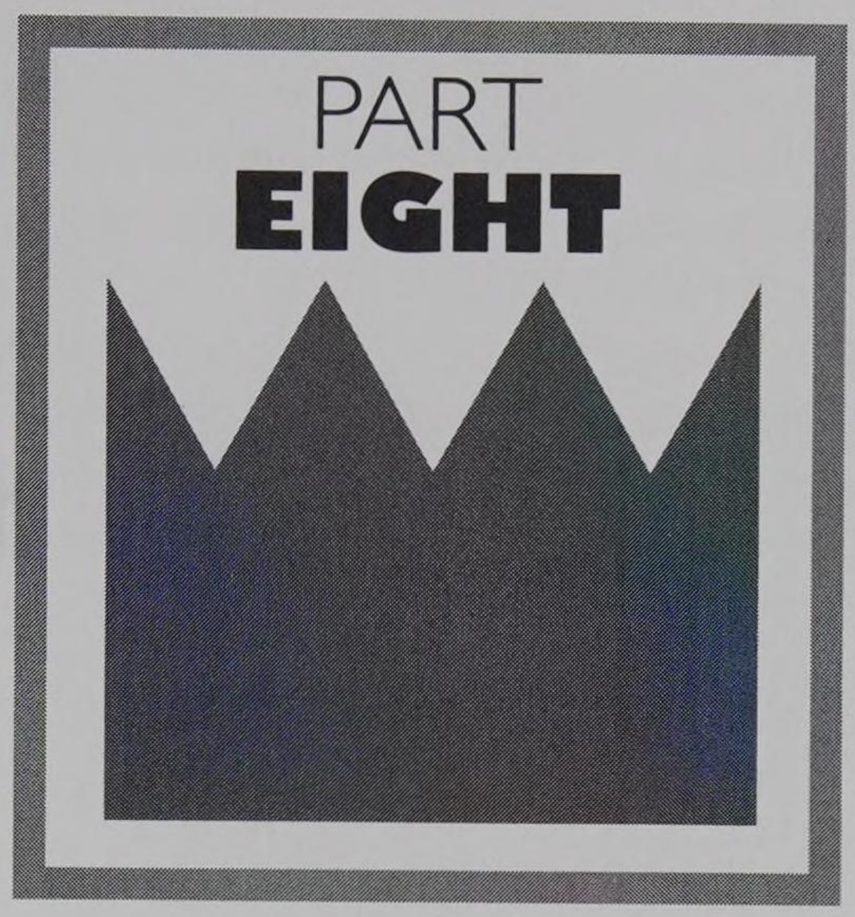
```
%%BeginPreview: 3 3 8 3
%%ff ff ff
%%ff 00 ff
%%ff ff ff
%%EndPreview
```

Following is a small program to read hex preview data and reconstruct a PGM image from it. We assume that the calling program has already read the Begin-Preview line and passes the width, height, and bits per pixel as arguments.

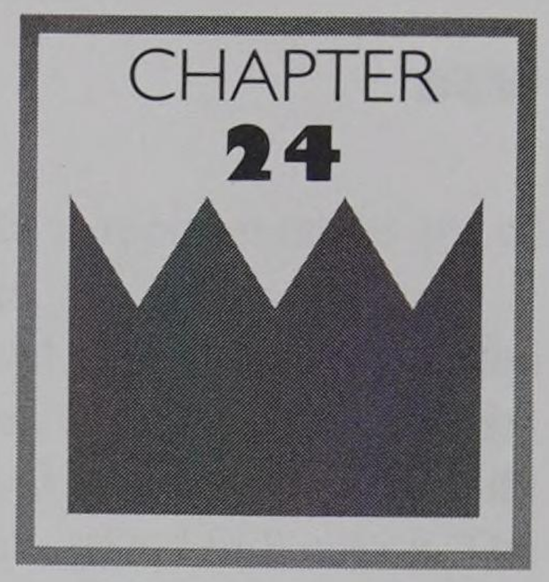
```
/* psidtopgm.c - convert PostScript "image" data into a portable
 * graymap */

#include "pgm.h"

static int gethexit ARGS(( FILE* ifp ));

void
main( argc, argv )
int argc;
char* argv[];
{
    FILE* ifp;
    gray* grayrow;
    register gray* gP;
    int argn, row;
    register int col, val;
    int maxval;
    int rows, cols, bitspersample;
    char* usage = "<width> <height> <bits/sample> [imagedata]";

    pgm_init( &argc, argv );

    argn = 1;

    if ( argn + 3 > argc )
        pm_usage( usage );

    cols = atoi( argv[argn++] );
    rows = atoi( argv[argn++] );
    bitspersample = atoi( argv[argn++] );
    if ( cols <= 0 || rows <= 0 || bitspersample <= 0 )
        pm_usage( usage );

    if ( argn < argc )
        {
        ifp = pm_openr( argv[argn] );
        ++argn;
        }
    else
        ifp = stdin;

    if ( argn != argc )
        pm_usage( usage );
```

```
maxval = pm_bitstomaxval( bitspersample );
if ( maxval > PGM_MAXMAXVAL )
    pm_error(
        "bits/sample is too large - try reconfiguring with
         PGM_BIGGRAYS" );

pgm_writepgminit( stdout, cols, rows, (gray) maxval, 0 );
grayrow = pgm_allocrow( ( cols + 7 ) / 8 * 8 );
for ( row = 0; row < rows; ++row)
    {
    for ( col = 0, gP = grayrow; col < cols; )
        {
        val = gethexit( ifp ) << 4;
        val += gethexit( ifp );
        switch ( bitspersample )
        {
        case 1:
        *gP++ = val >> 7;
        *gP++ = ( val >> 6 ) & 0x1;
        *gP++ = ( val >> 5 ) & 0x1;
        *gP++ = ( val >> 4 ) & 0x1;
        *gP++ = ( val >> 3 ) & 0x1;
        *gP++ = ( val >> 2 ) & 0x1;
        *gP++ = ( val >> 1 ) & 0x1;
        *gP++ = val & 0x1;
        col += 8;
        break;

        case 2:
        *gP++ = val >> 6;
        *gP++ = ( val >> 4 ) & 0x3;
        *gP++ = ( val >> 2 ) & 0x3;
        *gP++ = val & 0x3;
        col += 4;
        break;

        case 4:
        *gP++ = val >> 4;
        *gP++ = val & 0xf;
        col += 2;
        break;

        case 8:
        *gP++ = val;
        ++col;
        break;

        default:
        pm_error( "bitspersample of %d not supported",
```

```
                    bitspersample );
                    }
                }
            pgm_writepgmrow( stdout, grayrow, cols, (gray) maxval, 0 );
            }
        pm_close( ifp );
        pm_close( stdout );

        exit( 0 );
    } /* main */

    static int
    gethexit( ifp )
    FILE* ifp;
        {
        register int i;
        register char c;

        for ( ; ; )
            {
            i = getc( ifp );
            if ( i == EOF )
                pm_error( "EOF / read error" );
            c = (char) i;
            if ( c >= '0' && c <= '9' )
                return c - '0';
            else if ( c >= 'A' && c <= 'F' )
                return c - 'A' + 10;
            else if ( c >= 'a' && c <= 'f' )
                return c - 'a' + 10;
            /* Else ignore -- whitespace. */
            }
    }
```




# Format Conversion




# Bitmap to Bitmap: The PBM Approach

Converting from one pixel format to another is generally straightforward, since one rectangular array of pixels is fundamentally like another. In general, one can read one file format, extract the array of pixels, and then write the same pixels in any other format.

## 24.1 USING PBM PROGRAMS TOGETHER

The PBM utilities were specifically designed to support transformations from one format to another. Since all of the utilities read or write PBM, PGM, or PPM formats, by plugging the utilities together one can construct a translator from any pixel format to any other. On Unix systems, the command language lets one "pipe" commands together and directly connect the output of one program to the input of another. For example, to turn a GIF image into a PCX image, one could say:

```
giftoppm input.ppm | ppmtopcx > output.pcx
```

MS-DOS users can use similar syntax to get the same effect. The only practical difference is that instead of using special pipe files, DOS puts the intermediate PPM data into a file, so the local disk has to have enough room for the intermediate file.

PBM also supports automatic promotion of file types. To translate a single-bit GEM IMG file to GIF, merely pass the output of gemtopbm to the input of ppmtogif, since the PPM libraries automatically read and convert PBM data.

## 24.2 BITMAP TRANSFORMATIONS

Promoting from a less expressive format to a more expressive format is easy, since there is no issue of how to treat the data—a white pixel remains white, a

50 percent gray remains a 50 percent gray, and so forth. Converting in the other direction, from a more expressive format to a less expressive one is more difficult. The goal is to produce the best-looking image possible given the limitations of the new format.

The topic of image transformations doesn't have anything to do with file processing per se, so because of space limitations, we only discuss the possible transformations in general terms here. The companion diskettes do contain programs that perform all of these transformations on PPM, PGM, and PBM files.

## Color to Gray Conversion

Converting from full color to gray is the simplest of these transformations. For each pixel, one need only determine the pixel's luminance, a value conventionally computed from the component values as:

$$L = (0.299 * R + 0.587 * G + 0.114 * B)$$

This formula is based on the eye's relative sensitivity to different colors. The heart of a program to turn colored pixels into gray is:

```
/* we have ifp: the input file pointer, cols and rows: the size
 * of the image, maxval and format: usual PPM values

    /* prepare to read input PPM file */
ppm_readppminit( ifp, &cols, &rows, &maxval, &format );

    /* prepare to write output PGM file */
pgm_writepgminit( stdout, cols, rows, maxval, 0 );

    /* buffer a row at a time */
pixelrow = ppm_allocrow( cols );
grayrow = pgm_allocrow( cols );

for ( row = 0; row < rows; ++row ) {
    ppm_readppmrow( ifp, pixelrow, cols, maxval, format );
    for ( col = 0; col < cols; ++col )
        grayrow[col] =
            ( 0.299 * PPM_GETR(pixelrow[col]) +
            0.587 * PPM_GETG(pixelrow[col]) +
            0.114 * PPM_GETB(pixelrow[col]) );

    pgm_writepgmrow( stdout, grayrow, cols, maxval, 0 );
}
```

## Color Quantization

Sometimes one has a color image with more colors than the local hardware can handle, such as a full-color image to be displayed on a screen that can only show

64 or 256 colors. A process called *quantization* selects a representative set of colors and then assigns one of those colors to each pixel.

For example, Figure 24.1 is a 551 by 579 pixel digitized photograph of a kitten using 246 gray scale values. To display this many colors, a screen needs at least 8 bits of screen memory per pixel. If the screen had only 3 bits per pixel, it could only represent 8 colors. Figure 24.2 is the same picture quantized to 8 colors. The image is much coarser but still recognizable.

**Figure 24.1**
Grayscale image with 246 gray values.

## Dithering

Quantizing alone produces adequate results; but, particularly in areas with gradual changes of color, the limited number of colors is quite noticeable. One way to decrease this effect is by *dithering,* spreading the quantization error around from pixel to pixel to avoid unwanted step effects. Figure 24.3 still uses only 8 colors but it also uses dithering. The image is much smoother. We show dithering here on grayscale images because this page is printed with only one color of ink, but dithering is equally applicable to color images.

The ultimate color reduction is to plain black and white, 1 bit per pixel. In this case dithering is essential to produce a useful image. Simple-minded quantization of the same picture produces the image in Figure 24.4, which is almost unrecognizable. But when the same conversion also uses dithering, the result is Figure 24.5, which is quite recognizable as the same picture.

For more information on color reduction, quantization, and dithering, see the References at the back of this book.

**Figure 24.2**
Grayscale image with 8 gray values.

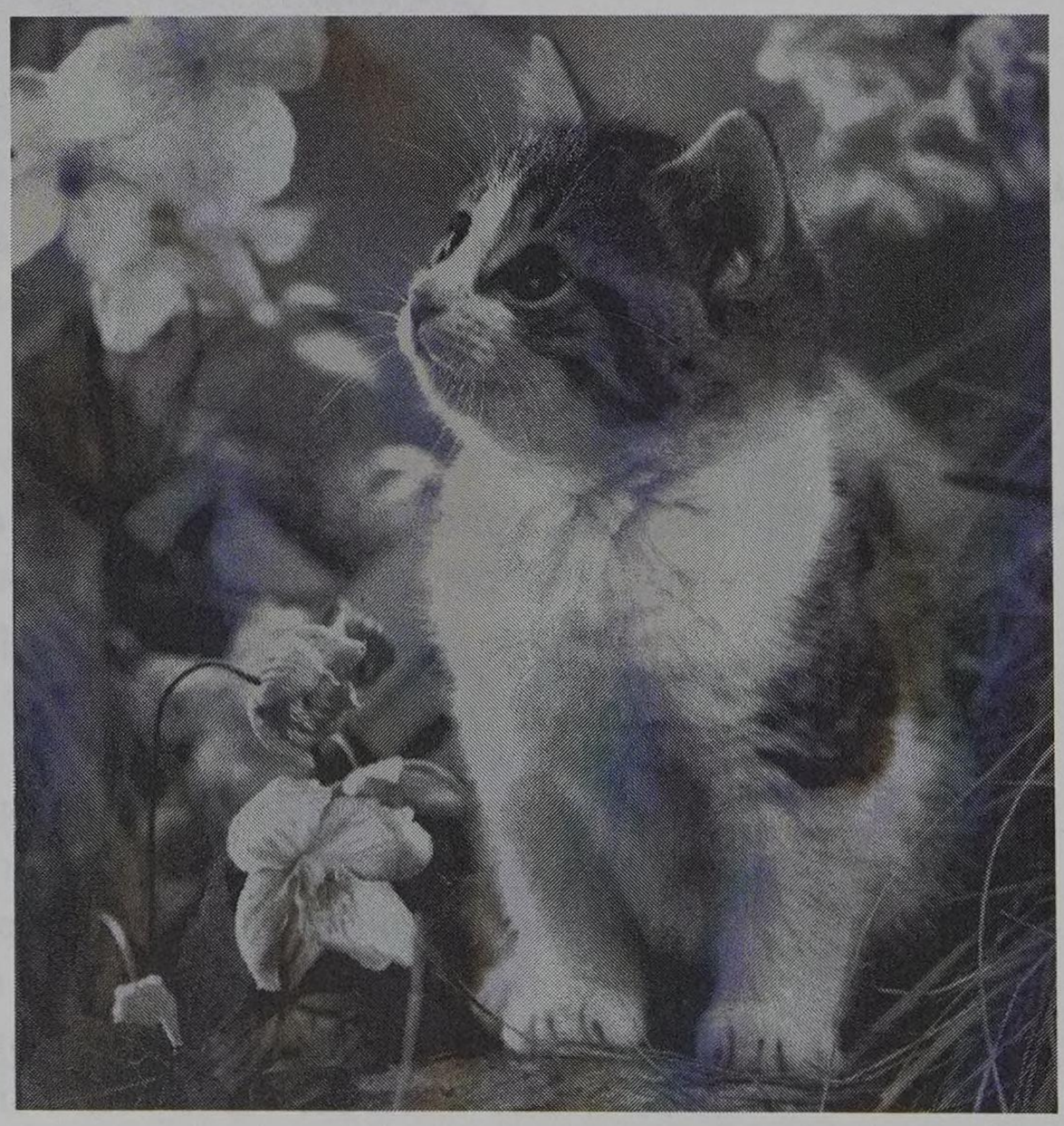


**Figure 24.3**
Grayscale image with 8 gray values, dithered.

**Figure 24.4**
Black-and-white quantized image.

**Figure 24.5**
Black-and-white dithered image.

CHAPTER
25

# Vector to Bitmap: Rasterization

Converting from vector to bit-mapped images is a surprisingly old problem. Early pen plotters used continuous paper on a drum, with stepping motors that drove the pen back and forth across the paper and the drum along the length of the paper. Determining the correct sequence of steps to draw a line on such a plotter turns out to be essentially the same problem as turning a line into a raster representation, since the set of possible step positions on the plotter really defined a raster array. The first algorithm for line rasterization was published by Bresenham in 1965, and variations have been used in one form or another ever since.

Here we first develop a small C PPM-based library that turns lines and circles into PPM pixel arrays. Then we splice that library into the C++ HP-GL reader from Chapter 20 to produce a working program that turns HP-GL into PPM.

## 25.1 RASTERIZATION LIBRARY

The PPM routines contain a simple but flexible rasterization library. It contains routines to draw points, lines, and circles. More advanced routines, not shown here, that draw splines and filled areas, are on the companion disk.

Each of the drawing routines takes a *drawproc,* which is a routine called at each point on the line or other object being drawn, or as an argument. The simplest thing for a drawproc to do is to set the pixel at the given position to the color of the object being drawn; but with different drawprocs, one can get different effects. For example, to draw a thicker line, the drawproc can set a $3 \times 3$ block of pixels around each point. The area-filling routines use a specialized drawproc (called from the usual line-, circle-, or other drawing routines) that builds up a data structure representing the outline of the area to be filled. Each drawproc takes a consistent set of arguments: the pixel array, the number of rows and columns in it, the maximum pixel values, the $x$ and $y$ coordinates of the point to draw, and a *clientdata* argument that the drawproc can interpret in any way needed. Typically

it is a pointer to a pixel of the color to be drawn, or a pointer to a data structure maintained by the drawproc.

To speed up the simple, most common case, the drawing routines check for a null drawproc and just draw a dot at the (*x*,*y*) position.

The header file declarations follow.

```
/* ppmdraw.h -- header file for simple drawing routines in libppm
*/
void ppmd_point_drawproc ARGS(( pixel** pixels, int cols, int rows,
        pixval maxval, int x, int y, char* clientdata ));

#define PPMD_NULLDRAWPROC (void (*)(pixel**, int, int, pixval, \
        int, int, char*)) 0
```

The line-drawing routine uses two static flags. The line-type flag, set by ppmd_setlinetype( ), can be NORMAL or NODIAGS. The latter makes sure that each point on the line is adjacent to another vertically or horizontally, not just diagonally, which can look better in some cases.

Normally, the routines clip all lines drawn to the borders of the pixmap. In some cases it makes more sense to let the drawproc do the clipping, in which case ppmd_setlineclipping( ) can turn the clipping flag off.

```
/* Outline drawing routines.  Lines, splines, circles, etc. */

int ppmd_setlinetype( int type );
#define PPMD_LINETYPE_NORMAL 0
#define PPMD_LINETYPE_NODIAGS 1

int ppmd_setlineclipping ARGS(( int clip ));
```

The two actual drawing routines we examine, ppmd_line( ) and ppmd_circle( ), each take as arguments the pixel array, size and maxval of the pixels, the drawproc, and the clientdata. For line drawing, the ends of the lines are given as (*x*0,*y*0) and (*x*1,*y*1). Circles are specified as the center and the radius.

```
void ppmd_line( pixel** pixels, int cols, int rows, pixval maxval,
        int x0, int y0, int x1, int y1,
        void (*drawprocP)(pixel**, int, int, pixval, int, int,
        char*),
        char* clientdata );

void ppmd_circle( pixel** pixels, int cols, int rows, pixval maxval,
        int cx, int cy, int radius,
        void (*drawprocP)(pixel**, int, int, pixval, int, int,
        char*),
        char* clientdata );
```

The drawing code starts with a simple drawproc that draws a point. In fact, this is the same effect one gets with a null drawproc, but we include it for completeness. It also turns out to be useful in the circle routine, which doesn't do its own clipping.

The line-type and clipping routines store their arguments in static variables and return the old values.

```
/* libppm5.c - ppm utility library part 5
*/

#include "ppm.h"
#include "ppmdraw.h"

#define DDA_SCALE 8192L
                    /* scale factor for fixed-point arithmetic */

void
ppmd_point_drawproc( pixel** pixels, int cols, int rows,
        pixval maxval,  int x, int y, char* clientdata )
{
    if ( x >= 0 && x < cols && y >= 0 && y < rows )
        pixels[y][x] = *( (pixel*) clientdata );
}

/* Outline drawing stuff. */

static int ppmd_linetype = PPMD_LINETYPE_NORMAL;

int
ppmd_setlinetype( type )
    int type;
    {
    int old;

    old = ppmd_linetype;
    ppmd_linetype = type;
    return old;
    }

static int ppmd_lineclip = 1;

int
ppmd_setlineclip( clip )
    int clip;
    {
    int old;

    old = ppmd_lineclip;
    ppmd_lineclip = clip;
```

```
    return old;
    }
```

The majority of the work in the line-drawing routine turns out to be the clipping.

First we check for the special case in which the line is really a point, and we handle that.

```
void
ppmd_line( pixel** pixels, int cols, int rows, pixval maxval,
int x0, int y0, int x1, int y1, void (*drawprocP)(pixel**, int,
int, pixval, int, int, char*), char* clientdata )
{
    register int cx0, cy0, cx1, cy1;

/* Special case zero-length lines. */
if ( x0 == x1 && y0 == y1 )
    {
    if ( ( ! ppmd_lineclip ) ||
         ( x0 >= 0 && x0 < cols && y0 >= 0 && y0 < rows ) )
        if ( drawprocP == PPMD_NULLDRAWPROC )
            ppmd_point_drawproc(
                pixels, cols, rows, maxval, x0, y0,
                clientdata );
        else
            (*drawprocP)( pixels, cols, rows, maxval, x0, y0,
            clientdata );
    return;
    }
```

Now we do the clipping, which requires checking all four points against the minimum and maximum array values. When we're done, cx0 and cy0 have the beginning of the line, clipped within the boundaries, and cx1 and cy1 have the end.

```
/* Clip. */
cx0 = x0;
cy0 = y0;
cx1 = x1;
cy1 = y1;
if ( ppmd_lineclip )
    {
    if ( cx0 < 0 )
        {
        if ( cx1 < 0 ) return;
        cy0 = cy0 + ( cy1 - cy0 ) * ( -cx0 ) / ( cx1 - cx0 );
        cx0 = 0;
        }
    else if ( cx0 >= cols )
```

```
        {
        if ( cx1 >= cols ) return;
        cy0 = cy0 + ( cy1 - cy0 ) * ( cols - 1 - cx0 ) /
                ( cx1 - cx0 );
        cx0 = cols - 1;
        }
    if ( cy0 < 0 )
        {
        if ( cy1 < 0 ) return;
        cx0 = cx0 + ( cx1 - cx0 ) * ( -cy0 ) / ( cy1 - cy0 );
        cy0 = 0;
        }
    else if ( cy0 >= rows )
        {
        if ( cy1 >= rows ) return;
        cx0 = cx0 + ( cx1 - cx0 ) * ( rows - 1 - cy0 ) /
                                                ( cy1 - cy0 );
        cy0 = rows - 1;
        }
    if ( cx1 < 0 )
        {
        cy1 = cy1 + ( cy0 - cy1 ) * ( -cx1 ) / ( cx0 - cx1 );
        cx1 = 0;
        }
    else if ( cx1 >= cols )
        {
        cy1 = cy1 + ( cy0 - cy1 ) * ( cols - 1 - cx1 ) /
                                                ( cx0 - cx1 );
        cx1 = cols - 1;
        }
    if ( cy1 < 0 )
        {
        cx1 = cx1 + ( cx0 - cx1 ) * ( -cy1 ) / ( cy0 - cy1 );
        cy1 = 0;
        }
    else if ( cy1 >= rows )
        {
        cx1 = cx1 + ( cx0 - cx1 ) * ( rows - 1 - cy1 ) /
                                                ( cy0 - cy1 );
        cy1 = rows - 1;
        }
```

Clipping might have shrunk the line to a point again, so we check again.

```
    /* Check again for zero-length lines. */
    if ( cx0 == cx1 && cy0 == cy1 )
        {
        if ( drawprocP == PPMD_NULLDRAWPROC )
            ppmd_point_drawproc(
```

```
                    pixels, cols, rows, maxval, cx0, cy0,
                    clientdata );
            else
                (*drawprocP)(
                    pixels, cols, rows, maxval, cx0, cy0,
                    clientdata );
            return;
            }
        }
```

There are two versions of the line-drawing loop, depending on whether the $x$ or $y$ distance is greater. The first version is for when the $x$ distance is greater. We move from cx0 to cx1, one pixel at a time. We also keep dy, a scaled version of the ratio of the $y$ distance to the $x$ distance, and srow, a scaled version of the $y$ position. For each $x$ pixel moved, we add dy to srow to get the new scaled $y$ position and divide that down by the scale factor to get row, the actual $y$ position. Then we plot the new position, adding an extra pixel above or below this one if necessary to prevent a diagonal move. (Bresenham's scheme is actually simpler, but this one is easier to explain.)

```
/* Draw, using a simple DDA. */
if ( abs( cx1 - cx0 ) > abs( cy1 - cy0 ) )
    { /* Loop over x domain. */
    register long dy, srow;
    register long dx, col, row, prevrow;

    if ( cx1 > cx0 )
        dx = 1;
    else
        dx = -1;
    dy = ( cy1 - cy0 ) * DDA_SCALE / abs( cx1 - cx0 );
    prevrow = row = cy0;
    srow = row * DDA_SCALE + DDA_SCALE / 2;
    col = cx0;
    for ( ; ; )
        {
        if ( ppmd_linetype == PPMD_LINETYPE_NODIAGS
           && row != prevrow )
            {
            if ( drawprocP == PPMD_NULLDRAWPROC )
                pixels[prevrow][col] = *( (pixel*) clientdata );
            else
                (*drawprocP)(
                    pixels, cols, rows, maxval, col, prevrow,
                    clientdata );
            prevrow = row;
            }
        if ( drawprocP == PPMD_NULLDRAWPROC )
```

```
                pixels[row][col] = *( (pixel*) clientdata );
            else
                (*drawprocP)(
                    pixels, cols, rows, maxval, col, row,
                    clientdata );
            if ( col == cx1 )
                break;
            srow += dy;
            row = srow / DDA_SCALE;
            col += dx;
            }
        }
```

If the *y* distance is greater than the *x* distance, we do the same thing, with the *x* and *y* roles reversed.

```
    else
        { /* Loop over y domain. */
        register long dx, scol;
        register long dy, col, row, prevcol;

        if ( cy1 > cy0 )
            dy = 1;
        else
            dy = -1;
        dx = ( cx1 - cx0 ) * DDA_SCALE / abs( cy1 - cy0 );
        row = cy0;
        prevcol = col = cx0;
        scol = col * DDA_SCALE + DDA_SCALE / 2;
        for ( ; ; )
            {
            if ( ppmd_linetype == PPMD_LINETYPE_NODIAGS
                                  && col != prevcol )
                {
                if ( drawprocP == PPMD_NULLDRAWPROC )
                    pixels[row][prevcol] = *( (pixel*) clientdata );
                else
                    (*drawprocP)(
                        pixels, cols, rows, maxval, prevcol, row,
                        clientdata );
                prevcol = col;
                }
            if ( drawprocP == PPMD_NULLDRAWPROC )
                pixels[row][col] = *( (pixel*) clientdata );
            else
                (*drawprocP)(
                    pixels, cols, rows, maxval, col, row,
                    clientdata );
            if ( row == cy1 )
```

```
                break;
            row += dy;
            scol += dx;
            col = scol / DDA_SCALE;
            }
        }
} /* ppmd_line */
```

Drawing a circle is, if anything, easier than drawing a line because the clipping issues are easier. We assume either that we know the entire circle is in the visible area or that the drawproc does the clipping as ppmd_point_drawproc( ) does.

Here we keep scaled versions of the *x* and *y* positions, called sx and sy, and a scaled version of the radius called e.

Then we start at the right side of the circle and generate each point from the last. We update s*x* by adding e*sy, scaled, and update sy by subtracting e*sx, also scaled. For readers who remember their calculus, this is essentially a numerical integration. The *x* and *y* values are the cosine and sine of the angle, from the horizontal to the current point, times a scale value. Since the cosine is the derivative of the sine, while the −sine is the derivative of the cosine, for each point we are taking a numerical integration step, adding in the derivative to get the next point.

When the circle comes all the way around to the starting point, we know that we have finished.

```
void
ppmd_circle( pixel** pixels, int cols, int rows,
        pixval maxval, int cx, int cy, int radius,
        void (*drawprocP)(pixel**, int, int, pixval, int, int,
        char*),
        char* clientdata )
{
    register int x0, y0, x, y, prevx, prevy, nopointsyet;
    register long sx, sy, e;

    x0 = x = radius;
    y0 = y = 0;
    sx = x * DDA_SCALE + DDA_SCALE / 2;
    sy = y * DDA_SCALE + DDA_SCALE / 2;
    e = DDA_SCALE / radius;
    if ( drawprocP == PPMD_NULLDRAWPROC )
        pixels[y + cy][x + cx] = *( (pixel*) clientdata );
    else
        (*drawprocP)( pixels, cols, rows, maxval, x + cx, y + cy,
         clientdata );
    nopointsyet = 1;
    do
        {
        prevx = x;
```

```
prevy = y;
sx += e * sy / DDA_SCALE;
sy -= e * sx / DDA_SCALE;
x = sx / DDA_SCALE;
y = sy / DDA_SCALE;
if ( x != prevx || y != prevy )
    {
    nopointsyet = 0;
    if ( drawprocP == PPMD_NULLDRAWPROC )
        pixels[y + cy][x + cx] = *( (pixel*) clientdata );
    else
        (*drawprocP)(
            pixels, cols, rows, maxval, x + cx, y + cy,
            clientdata );
    }
while ( nopointsyet || x != x0 || y != y0 );
}
```

## 25.2 HPGL TO PPM

We now add some calls to the drawing routines to the C++ HP-GL reader from Chapter 20 to get a program to convert HP-GL to PPM.

We start with the same code as we saw in Chapter 20, with the additions of some #include lines to get the declarations of the PPM datatypes and routines.

```
// gltoppm.cpp
// Read and interpret basic HP-GL and produce a PPM file
// John Levine, 4/93

// get declarations for PPM draw routines
extern "C" {
#define __STDC__ p 1          /* Need this to get prototypes */
#include "ppm.h"
#include "ppmdraw.h"
}

#include "hpglfile.h"
#include <fstream.h>
#include <string.h>

float xmin, ymin, xmax, ymax;   // plot limits
float xoffset = 0, xfactor = 1.0;       // scaling info
float yoffset = 0, yfactor = 1.0;

const int xlimit = 200, ylimit = 200;       // size of actual page

// scale routines produce ints, for PPM draw arguments
```

```
inline int xscale(float x) { return (int)((x-xoffset)*xfactor); }
inline int yscale(float y) { return (int)((y-yoffset)*yfactor); }
```

We simulate HP-GL pens 0 through 4 with five different pixel values in color_pixels[ ] representing different pen colors.

```
// color array
pixel color_pixels[5];  // will be set up later
const pixval maxval = PPM_MAXMAXVAL;

int
main(int ac, char **av)
{
    float arg;
    float xarg, yarg;
    int cmdcode;
    int dodraw = 0;          // is the pen down?
    int dorel = 0;           // are we moving relatively?
    float curx = 0,          // current pen position
        cury = 0;
    int color = 1;           // current pen color
    pixel   **image;         // PPM image data
    pixel blackpix, whitepix;
    int x, y;

    ppm_init( &ac, av );
    ifstream fin(av[1]);     // read from input file
    hpglfile hf(fin);
```

We allocate a PPM image in the usual way. In this simple routine, the size of the PPM image is fixed. More realistically the size would be a parameter to the program.

Then we set up the color array to five common colors, and we initialize the array to all white. (Plotters usually draw black on white, since black pens and white paper are easier to come by than the converse.)

```
    // set up our PPM image
    if ((image = ppm_allocarray(xlimit, xlimit)) == NULL)
        pm_error("couldn't alloc space for image" );

    PPM_ASSIGN(blackpix, 0, 0, 0);
    PPM_ASSIGN(whitepix, maxval, maxval, maxval);
    // set up list of colors
    color_pixels[0] = whitepix;          // color 0 is white
    color_pixels[1] = blackpix;          // color 1 is black
    PPM_ASSIGN(color_pixels[2], maxval, 0, 0);  // 2 is red
    PPM_ASSIGN(color_pixels[3], 0, maxval, 0);  // 3 is green
    PPM_ASSIGN(color_pixels[4], 0, 0, maxval);  // 4 is blue
```

```
// initialize page to white
for(x = 0; x < xlimit; x++)
    for(y = 0; y < ylimit; y++)
        image[y][x] = whitepix;
```

The main loop reads the HP-GL commands as before and adds calls to ppmd_circle to handle CI commands and to ppmd_line to handle the various line-drawing commands.

```
    while(hf.getcmd(cmdcode)) {

    switch(cmdcode) {

default:
    cerr << "Mystery code" << cmdcode << endl;
    return 1;

    // set scale
case hpglfile::SC:
    hf.getnumber(xmin); hf.getnumber(xmax);
    hf.getnumber(ymin); hf.getnumber(ymax);

    if(xmin == xmax || ymin == ymax) {
        cerr << "Zero scaling error\n";
        return 1;
    }

    // translate to offset and scale
    // to 0..xlimit in both dimensions
    xoffset = xmin; yoffset = ymin;
    xfactor = xlimit/(xmax-xmin);
    yfactor = ylimit/(ymax-xmin);
    continue;

case hpglfile::SP:
    hf.getnumber(color);
    continue;

case hpglfile::PG:  // Page eject, end of plot
    goto last;    // two-level break, simulate with a goto

    // draw a circle
case hpglfile::CI:
    xarg = hf.getnumber(xarg);
    ppmd_circle(image, xlimit, ylimit, maxval, xscale(curx),
        yscale(cury), (int)(xarg*xfactor), PPMD_NULLDRAWPROC,
        (char*)&color_pixels[color]);
```

```
            cerr << "Circle of radius " << xarg << " at ("
                << curx << "," << cury << ") color "
                << color << endl;
            continue;

        case hpglfile::PA:  // Plot Absolute
            dorel = 0;
            break;      // fall into common draw code

        case hpglfile::PD:  // Pen Down
            dodraw = 1;
            break;      // fall into common draw code

        case hpglfile::PR:
            dorel = 1;
            break;      // fall into common draw code

        case hpglfile::PU:
            dodraw = 0;
            break;      // fall into common draw code
            }

            // common draw code for PA, PR, PU, PD
            while(hf.getxy(xarg, yarg)) {
                float oldx = curx;
                float oldy = cury;

                if(dorel) {
                    curx += xarg;
                    cury += yarg;
                } else {
                    curx = xarg;
                    cury = yarg;
                }
                if(dodraw) {
                    ppmd_line(image, xlimit, ylimit, maxval,
                        xscale(oldx), yscale(oldy),
                        xscale(curx), yscale(cury),
                        PPMD_NULLDRAWPROC, (char*)
                        &color_pixels[color]);

                    cerr << "draw from (" << xscale(oldx) << ","
                         << yscale(oldy)
                         << ") to (" << xscale(curx) << ","
                         << yscale(cury)
                         << ") color " << color << endl;
                }
            }
        } /* while */
```

When we get to the end of the HP-GL, we write out the PPM and we're done.

```
last:
    ppm_writeppm(stdout, image, xlimit, ylimit, maxval, 0 );
    pm_close(stdout);
    return 0;
} /* main */
```

## 25.3 EXTENDING THE RASTERIZER

The most important HP-GL feature that we do not support is text labels. The most practical way to treat text is to use line drawing rather than bit-mapped fonts and to treat each character as a series of line segments. HP-GL text can be drawn at arbitrary sizes and angles, and it is much easier to scale and rotate lines than it is bitmaps.

CHAPTER
26

# Guessing a File Format

A frequent problem in graphics file processing is determining the format of a particular file. The easiest but least reliable way to do so is to use the file's extension. On most systems, a PCX file name ends with .PCX, a TIFF file with .TIF or .TIFF and so forth. A more reliable technique is what's known as the magic number approach. Nearly all file formats have an identifiable byte string, either by deliberate design or by fortunate coincidence, at or near the beginning of the file.

A table of common types and magic numbers follows (Table 26.1). The length and offset are in decimal, with offset 0 being the beginning of the file. The magic numbers are written as pairs of characters, meaning hex byte values, or as single characters, meaning the literal ASCII characters, and are written in the actual order the bytes appear in the file.

**Table 26.1**
File Format Magic Numbers

| FORMAT | OFFSET | LENGTH | VALUE | COMMENTS |
|---|---|---|---|---|
| MacPaint | 0 | 4 | 00 00 00 02 | Version number field, not always set correctly. |
| PCX | 0 | 1 | 0a | |
| GEM IMG | 0 | 4 | 00 01 00 08 | Fourth byte may also be 09. |
| IFF/ILBM | 0 | 4 | F O R M | |
| BMP | 0 | 2 | B M | |
| Targa | 0 | 1 | 00 | See note below. |
| GIF 87 | 0 | 6 | G I F 8 7 a | GIF files without extensions. (Even in the GIF 89 spec, files with no GIF89 features are supposed to have GIF87a headers.) |
| GIF 89 | 0 | 6 | G I F 8 9 a | GIF files with GIF89 extensions. |
| JFIF | 0 | 11 | FF D8 FF E0 xx xx 4A 46 49 46 00 | The xx bytes vary from file to file. |
| HP-GL | | | | No magic number, but files usually start with an HP-GL command, which is two uppercase letters followed by a digit or semicolon. |
| | 0 | 4 | 1B % 0 A | HP-GL to be printed on a PCL5 printer starts with this string to switch to HP-GL mode. The 0 is sometimes replaced by 1, 2, or 3. |
| WMF | 0 | 6 | 01 00 09 00 00 03 | The 9 is the header size, which may in theory be larger if the header is extended. The 03 is the Windows version number, 02 for old files from Windows 2. |
| | 0 | 4 | D7 CD C6 9A | Placeable WMF header |
| PCL | 0 | 2 | 1B E | See note below. |
| PostScript | 0 | 2 | % ! | Header string not mandatory, but found in nearly all files. |
| PBM | 0 | 2 | P 1 | P 4 for raw PBM |
| PGM | 0 | 2 | P 2 | P 5 for raw PGM |
| PPM | 0 | 2 | P 3 | P 6 for raw PPM |

Note on Targa files: There is no magic number at the front. The first thing in the file is the length of the image ID, which is almost always zero, since most files don't have an ID. Targa 2.0 files have an identifying string at the end of the file, which is "TRUEVISION-TARGA" followed by a period and a byte of binary zeros.

Note on PCL files: There is no magic number, but most begin with this printer reset sequence. Some begin with a longer sequence starting with 1B and an asterisk.

# References

David C. Kay and John R. Levine, **Graphics File Formats**, Windcrest, 1992.
Describes a wide variety of graphics file formats, including all of the ones used in this book, most of them to the bit level.

Adobe Systems, **PostScript Language Reference Manual**, second edition, Addison-Wesley, 1990.
The definitive reference for PostScript, includes conventions for EPS and EPSI files.

**TIFF Revision 6.0**, Aldus Corp., June 1992, available in PostScript on most BBSes or from Aldus at +1-206-628-6593.
The most recent and most complete specification for TIFF format.

Gregory K. Wallace, *The JPEG Still Picture Compression Standard*, "Communications of the Association for Computing Machinery," 34(4):30–44, 1991.
A clear and informative overview of JPEG.

William B. Pennebaker and Joan L. Mitchell, **JPEG Still Image Data Compression Standard**, Van Nostrand Reinhold, 1993.
The best book available on JPEG, including the complete text of the draft ISO JPEG standards.

J. E. Bresenham, *Algorithm for Computer Control of a Digital Plotter*, "**IBM Systems Journal**," 4:1, July 1965, pp. 25–30.
The original reference on rasterizing lines.

# About the Software

## WHAT IS ON THIS DISK?

This book's Companion Disk contains all of the source code in the book as well as thousands of lines of other related code not in the book. The code is in three large libraries:

PBMPLUS The Poskanzer Bitmap utilities, which read and write several dozen different bitmapped formats and also include file transformations such as scaling, color quantizing, and dithering.

TIFF The Leffler TIFF library, version 3.0. The most complete and accurate TIFF implementation available. It implements the full TIFF 6.0 spec (except for JPEG-in-TIFF, which has yet to be defined in a way that can be implemented) with some popular but nonstandard extensions.

JPEG The Independent JPEG Group software, version 4. Complete JPEG reading and writing libraries, routines to read and write GIF, TARGA, PBMPLUS, and JIFF (the most common JPEG file format), and driver code to build standalone translators between JFIF and the three other formats.

## SYSTEM REQUIREMENTS

A system with an ASCII C compiler. The code has been tested on Microsoft C 7.0 and numerous UNIX compilers, and should be easily portable to other 16- and 32-bit C systems. Refer to your compiler's documentation for information on MAKEFILEs.

## MAKING A BACKUP COPY

Before you start to use the enclosed disk, we strongly recommend that you make a backup copy of the original. Remember, however, that a backup disk is for your

own personal use only. Any other use of the backup disk violates copyright law. Please take the time now to make the backup, using the instructions below:

1. Insert your DOS disk into drive "A" of your computer, (assuming your floppy drive is "A").

2. At the `A:>`, type `DISKCOPY A: A:` and press Return.

   You will be prompted by DOS to place the disk to be copied into drive A.

3. Place the Programming for Graphics Files disk into drive A.

Follow the directions on the screen to complete the copy. When you are through, remove the new copy of the disk and label it immediately. Remove the original disk and store it in a safe place.

## INSTALLING THE DISK

The enclosed diskette contains 636 individual files in compressed format. In order to use the files, you must run the installation program from the diskette.

You can install the diskette onto your computer by following these simple steps:

1. Assuming your floppy drive is "A," insert the Programming for Graphics Files disk into your computer's "A" drive.

2. At the `A:>` type `INSTALL` and press Return.

The installation program will be loaded. After the title screen appears, you will be given the options shown in Figure 1.

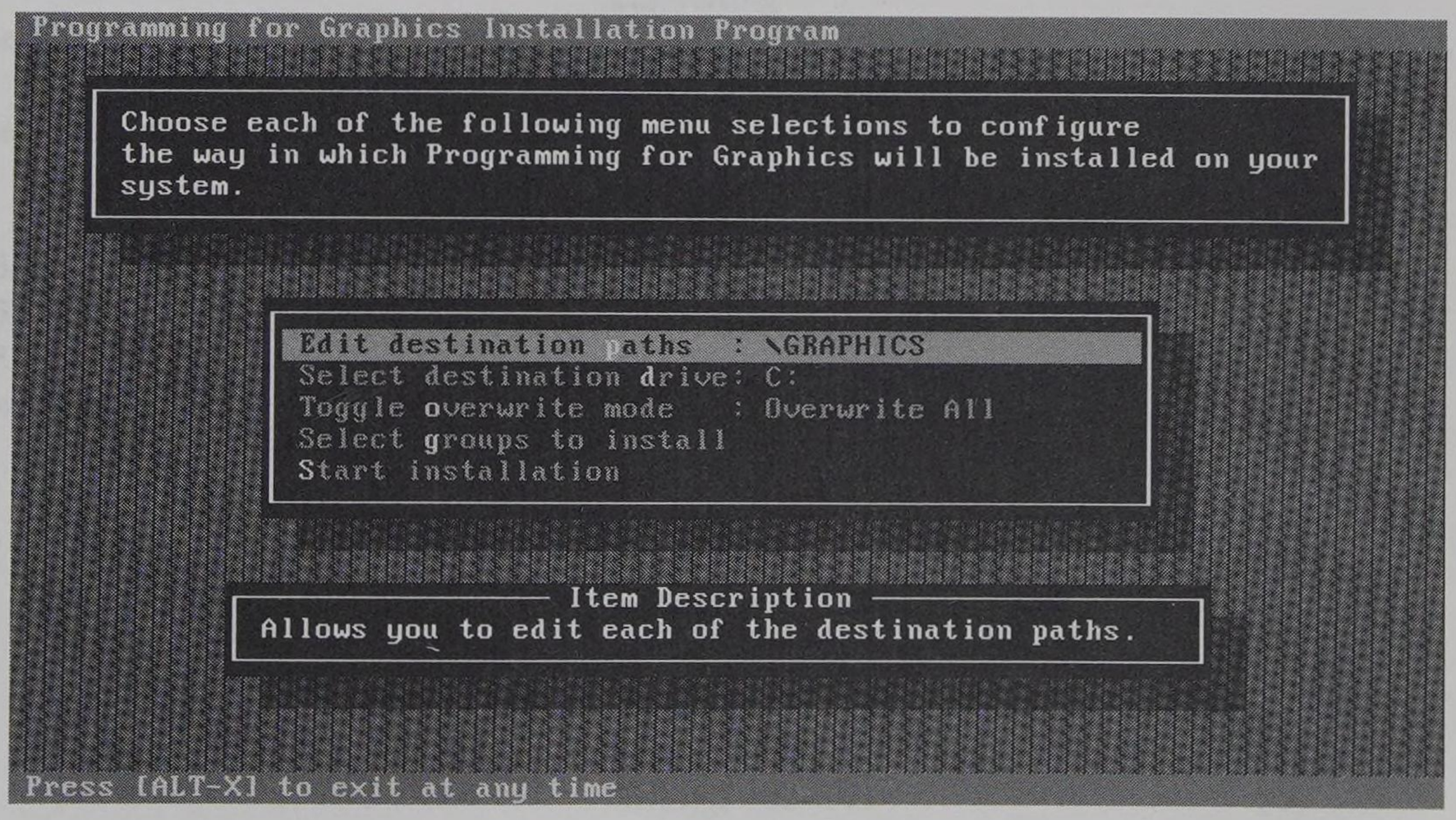



**Figure 1**
Programming for Graphics Files installation program startup screen.

To change any of the default settings, type the highlighted letter or move the menu bar to the desired option and press Enter.

3. To start the installation, type "S" or move the menu bar to the **Start installation** option and press Enter.

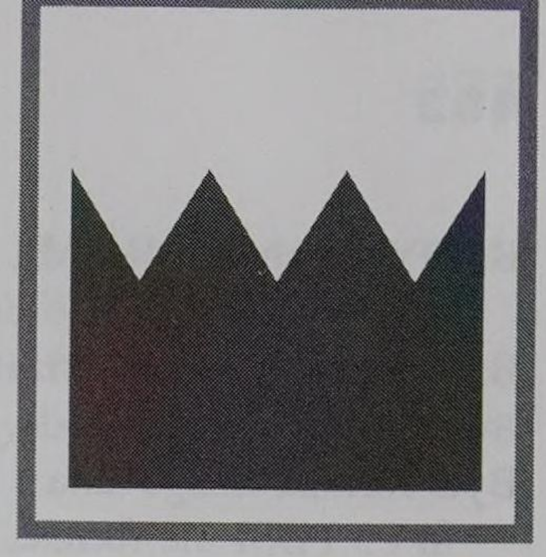
After the installation is complete, remove your original diskette and store it in a safe place.

## USER ASSISTANCE AND INFORMATION

John Wiley & Sons, Inc. is pleased to provide assistance to users of this package. Should you have questions regarding the use of this package, please call our technical support number at (212) 850-6194 weekdays between 9 am and 4 pm Eastern Standard Time.

To place additional orders or to request information about other Wiley products, please call (800) 879-4539.

# Index




**A**

abs definition, 41
Algorithms
  edge-detection, 13
  LZW compression, 231
  null compression techniques, 350
  rasterization, 12
    journal reference, 477
Amiga
  screen modes, handling, 159
  writing HAM mode ILBM files for, 166
ARGS definition, 41
Arithmetic coding, 19. *See also* Compression techniques
  JPEG use, 388
Arrays. *See* Memory management
ASCII format
  advantages and disadvantages as a graphics encoding format, 3
  image data
    in P1 PBM file format, 36
    in P2 PBM file format, 37
    in P3 PBM file format, 37
  PBM file, header designation, 43

**B**

Banding, 30
bcmp definition, 40
bcopy definition, 40
Big-endian byt e order, 55
  IFF numbers stored as, 154
  IMG use of, 143, 145
  long
    reading, 56
    writing, 57
  short, reading and writing, 56
  TIFF support of, 255
Binary format
  compressed file formats, advantages and disadvantages, 3
  image data, in P4 PBM file format, 36
  PBM file. *See* Raw files, PBM
Bit planes, 9
BITFIELDn source code, 268
Bitmap file format, 3, 7, 35–42. *See also* BMP file format; Formats
  converting
    to bitmap, 457–463
    vectors to, 462–474
  Postscript handling of images in, 11
  transformations of, 457
  Windows, 179–191
Bitn source code, 268
BitSet source code, 239
bitspersample source code, 442
BMHD subchunk (ILBM), 154
  parsing, 157
  writing, 169
BMP file format, 179–191. *See also* Bitmap file format; Formats
  decoding, 181
  encoding, 188
  reading, 180–185
  writing, 185–191
bmp.h source code, 179
bmptoppm.c source code, 180

BODY subchunk (ILBM), 154
writing, 170
Bresenham, basic rasterization algorithm, 12
BumpPixel source code, 222
Byte format, image data
in P5 PBM file format, 37
in P6 PBM file format, 37
bzero definition, 40

C

c_ui_method_selection source code, 393
CAMG subchunk (ILBM), 154
parsing, 159
writing, 170
Case-insensitive, keyword matching, 48
CCITT RLE compression. *See* Compression techniques, CCITT RLE
char_init source code, 238
char_out source code, 238
CheckDirCount source code, 310
CheckMalloc source code, 299
Chrominance, 15, 387. *See also* Luminance
Circles, rasterization routines, 469–470
cjpeg program, 389
cl_block source code, 236
cl_hash source code, 237
CleanupField source code, 295
Clipping routines, 463–467
CMAP subchunk (ILBM), 154
parsing, 158
writing, 169
Color, 14–16. *See also* Grayscale
24-bit, writing Targa files, 203–214
bitmap images, GIF format handling of, 217–254
coding, in JPEG encoding, 387
converting to gray scale, 458
full-color images, 79–101
checking for in ILBM format files, 160
handling with Targa file format, 192–214
processing in ILBM format files, 163
TIFF class that supports, 255
hash table, 93, 100, 208
histogram, 92, 99
maps, 15, 92–101
checking for in ILBM format files, 160
computing BMP, 187
computing GIF, 220
converting PPM format to PCX format, 133
creating, 94
handling Targa, 196
processing ILBM format, 168
reading PCX file, 124
TIFF class that supports, 255
writing BMP, 189
writing GIF, 227
writing ILBM, 169
models, 14–15
reduction, 458
true-color images
libraries for, 79–101
PCX format supporting, 142
scanned, handling with Targa file format, 192–214
Windows BMP format support of, 179
colorhash_table , 93
colorhist_item , 92, 93
colorhist_list , 92
colorhist_list_item , 93
colorhist_vector , 92
colorstobpp source code, 173, 221
compress source code, 231
compress.c source code, 230
Compress_info_struct source code, 390
Compress_methods_struct source code, 390
Compression techniques
arithmetic coding, 19
CCITT RLE, 255, 360
dictionary
GIF, 217–254
JPEG, 385–403
TIFF, 255–384
GIF images , 246
Group 3 fax, 361
Group 4 fax, 361
Huffman coding, 18
IMG files, 143, 151
introduction, 17–20
JPEG, 361, 389–395
lossless, 19, 385
lossy, 10, 19, 385, 388
LZW dictionary, 18, 217, 228–238, 248, 361
NeXt, 361
null, algorithm and routines, 350
PackBits, 17–18, 118
ILBM handling of images compressed with, 161
MacPaint file format use of, 115
TIFF use of, 354–360
PCX format
images, 135
plane data, 140

Pixar RLE, 361
RLE, 201, 207, 212, 388
run-length
GEM IMG, 143–153
IFF/ILBM, 154–176
MacPaint, 115–121
PCX, 121–142
Silicon Graphics RLE, 361
Thunderscan, 361
TIFF techniques, 345–361
Compuserve, GIF format, 217–254
`compute_runlengths` source code, 212
`const` definition, 41
`cvtRational` source code, 312

**D**

`d_ui_method_selection` source code, 401
Data stream
GIF format, 217–254
JPEG as encoding standard for, 386
DCT (Discrete Cosine Transform), 387
`Decompress_info_struct` source code, 396
Decompression, JPEG. 395–404. *See also* Compression techniques
`defaultHandler` (`tif_error.c`) source code, 381
`defaultHandler` (`tif_warning.c`) source code, 382
`deref` source code, 28
Dictionary. *See also* Compression techniques, LZW dictionary
coding techniques, 18
compressed formats
GIF, 217–254
JPEG, 385–403
TIFF, 255–384
Digitized photographs
HAM mode use for, 155
JPEG handling of, 385
Dithering, 459
`djpeg` program, 389
`DoExtension` source code, 245
`DumpModeDecode` source code, 353
`DumpModeEncode` source code, 352
`DumpModeSeek` source code, 354
`dumpObject` source code, 425

**E**

Electronic Arts. *See* ILBM subformat (IFF)
EPS (Encapsulated PostScript), reading, 451
Error handling
PBM applications, 51
pm_error definition, 41
pm_message definition, 41
pm_perror definition, 42
pm_usage definition, 42
TIFF, 381
`error_exit` source code, 399
`EstimateStripByteCounts` source code, 309
Extended
memory, MS-DOS, virtual segmented pixel arrays in C++, 24–30
text formats, 11
`External_methods_struct` source code, 398
Extra half-bright mode, 154, 161

**F**

Files
closing
pm_close source code, 42
with pm_close, 55
encoding, BMP, 188
handling, by PBM programs, 54
large, memory management techniques, 23–30
opening
pm_openr source code, 42
pm_openw source code, 42
with pm_openr, 54
with pm_openw, 55
partial, memory management techniques, 30
plain PBM. *See also* ASCII format
distinguishing between raw PBM files and, 61
reading a single byte, 62
writing, 65–67
reading
BMP, 180–185
GIF, 239–254
HP-GL, 407
ILBM, 156–166
IMG, 143–148
location of code for, xi
MacPaint format, 116
PBM format, 59
PCX format, 122–131
PGM format, 71
plain PPM, 85

Files, reading *(continued)*
PNM format, 104–107
PPM format, 83
Targa, 193–203
small and medium, memory management issues, 21
types, converting among, 12
writing
BMP, 185–191
GIF, 218–238
ILBM, 166–175
IMG, 148–153
location of code for, xi
MacPaint format, 117
PCX format, 131–142
PGM file format, 74–78
PNM format, 107–111
PPM format, 88–9 2
Targa, 203–214
`filltemp` source code, 119
`findScheme` source code, 348
`flush_char` source code, 238
`forceplain` argument, 64
FORM
chunk, reading in, 157
ILBM header, writing, 169
Formats. *See also* Bitmap file format; BMP file format; GIF file format; HP-GL (Hewlet-Packard Graphics Language); ILBM subformat (IFF); IMG image format; JFIF; JPEG (Joint Photographic Exerts Group) format; MacPaint format; PBM format; PCL (HP Printer Control Language); PCX format; PGM format; PNM format; PPM format; Targa format; TIFF (Tagged Image File Format); WMF (Windows metafiles) format
conversion
bitmap to bitmap, 457–463
vector to bitmap, 462
file
categories of, 3
compression techniques, 17–20
converting among, 12
guessing, 475
location of code for reading and writing, xii
graphics
location of code for reading and writing files, xi
overview, 3
number, turning into a type number with PPM_FORMAT_TYPE macro, 81

## G

GEM windowing system, IMG image format, reading and writing, 143–153
`get_byte` source code, 165
`get_input_row` source code, 392
`get_map_entry` source code, 200
`get_pixel` source code, 201
`GetByte` source code, 131
`getbyte` source code, 203
`getcmd` source code, 409
`GetCode` source code, 246
`GetDataBlock` source code, 246
`getdword` source code, 423
`getfourchars` source code, 165
`getMode` source code, 337
`getnumber` source code, 411
`getpixel` source code, 22
`GetPixel` source code, 221
`getstring` source code, 413
`GetWord` source code, 131
`getword` source code, 423
GIF file format. *See also* Formats
encoding, 222
extension types, 244
handling, 217–254
LZW compression use by, 19
reading, 239–254
writing, 218–238
`GIFEncode` source code, 224
`GIFNextPixel` source code, 223
`giftoppm.c` source code, 239
`gltoppm.cpp` source code, 470
graph coordinates, 6
Graphics files, 3–6
types of, 7–14
Grayscale, 14. *See also* Color; Images
checking for, in ILBM format files, 160
libraries supporting, 68–78
PGM format overview, 36
Targa files, writing, 203–214
TIFF class that supports, 255
writing out colormap for HAM, 170
Group 3 fax compression, 361. *See also* Compression techniques
Group 4 fax compression, 361. *See also* Compression techniques

## H

HAM (Hold and Modify) mode
  checking for, 160
  processing pixels stored in, 164
Hash table, 98, 134
Headers
  BMP
    source code, 179
    writing, 189
  files, PBM routines source code, 38
  ILBM, writing, 169
  IMG
    reading, 145
    writing, 150
  PBM, reading, 60, 63, 149
  PCX, 121
    reading, 123
    writing, 136
  Targa
    include file describing, 192
    writing out, 209
  TIFF library, 257
`headers` source code, 120
Hold and Modify (HAM) mode, 154
HP-GL (Hewlett_Packard Graphics Language), 6, 407–417. *See also* Formats
  converting to PPM, 470
  files, reading, 407
  high-level reader, 414
  low-level reader, 408
`hpglfile` C++ class source code, 408
`hpglfile.h` source code, 408
Huffman coding, 18. *See also* Compression techniques
  JPEG use, 388

## I

IFD (Image File Directory), 298–335. *See also* TIFF (Tagged Image File Format)
  managing, source code, 276–298
  reading, 298–320
  structure and constant source code, 257–263
  writing, 320–335
IFF. *See* ILBM subformat (IFF)
ILBM subformat (IFF), 154–175. *See also* Formats
  files
    reading, 156–166
    writing, 166–175
  FORM chunk, parsing, 157
  header, 155
  subchunk types, 154
`ilbmtoppm.c` source code, 156–166
Images. *See also* Color; Grayscale; TIFF (Tagged Image File Format)
  computer-generated, GIF format handling of, 217–254
  geometry of, 5
  monochrome, 14, 43–67
  raster, 7
    ray-traced
    Postscript handling of, 11
    Targa handling of, 192–214
  scanned
    GIF format handling of, 217–254
    Postscript handling of, 11
    single-bit per pixel, libraries for, 43–67
IMG image format, 143–153. *See also* Formats
  reading, 143–148
  writing, 148–153
`Index` definition, 40
`input_init` source code, 391
`input_term` source code, 393
Integer base two logarithms, routines that perform, 53
Integers, reading and writing binary, 55
Interlacing, GIF images, 217
Interleaving
  in JPEG, 387
  rows, 9
    supported by Targa handling routines, 192
  Targa, handling, 198

## J

JFIF, formats, 386. *See also* Formats
JPEG (Joint Photographic Experts Group) format, 385–404. *See also* Formats
  compression, 361, 389–395
    virtual memory implications, 23
  concepts and use, 385
  decompression, 395–404
  format descriptions, 386
  library
    code characteristics and structures, 389
    components and use, 388–404
  library code, location, xii
  lossy compression use for pixel ordering, 10
Keyword, matching, case-insensitive, 48

## L

libpbm.h source code, 45
libpbml.c source code, 46
libpbm2.c source code, 59
libpgm.h source code, 70
libpgml.c source code, 70
libppm.h source code, 82
libppm4.c source code, 464
Lines, rasterization routines, 462–469
Little-endian byte order, 55
  GIF handling, 224
  long, reading and writing, 58
  short, reading and writing, 57
  swapping bytes in, 148
  Targa file use, 195
  TIFF support of, 255
LM_to_uint source code, 239
lockrow source code, 29
Lossless compression. *See* Compression techniques, lossless
Lossy compression. *See* Compression techniques, lossy
Luminance
  computing for a pixel, with PPM_LUMIN macro, 82
  luminance/chrominance color model, 15
LZW dictionary compression. *See* Compression techniques, LZW dictionary
LZWReadByte source code, 248

## M

machdep.h source code, 271
macp.h source code, 115
MacPaint format, 115–120. *See also* Formats
  PBM routines used for handling files from, 43
  reading, 116
  writing, 117
Magic numbers. *See also* PBM format, header
  BMP file, handling, 182
  determining file format with, 475
  GIF, 226
  PBM header file component, 43
  PCX format, 122
  PGM file format header component, 68
  PPM format, 80
  reading from PBM format file, 60
  reasons not defined for PNM format, 102
  TIFF, 255
max definition, 41
Memory management, 21–31
  allocation
    library routines supporting, 45
    pbm.h source code, 44
    with pm_allocrow, 46
  deallocation
    with pm_freearray, 47
    with pm_freerow, 46
  metafiles, 30
  MS-DOS environment, 23
  partial files, 30
  vector files, 30
Metafiles, 5, 10. *See also* WMF (Windows metafiles)
  memory management techniques, 30
min definition, 41
MissingRequired source code, 310
MS Windows
  Windows metafiles. *See* WMF (Windows metafiles)
  Windows Bitmap (BMP) format. *See* BMP file format
MS-DOS, memory management issues, 23

## N

NeXt compression, 361. *See also* Compression techniques

## O

odd definition, 41
OkToChangeTag source code, 288
output source code, 235
output_init source code, 399
output_term source code, 401
P1 type field, 36
P2 type field, 37
P3 type field, 37
P4 type field, 36
P5 type field, 37
P6 type field, 37

## P

PackBits compression. *See* Compression techniques, PackBits
PackBitsDecode source code, 359
PackBitsEncode source code, 357–359
PackBitsEncodeChunk source code, 356
PackBitsPreEncode source code, 355
packit source code, 118
Page description languages, 11

Palettes. *See* Color, maps
pbm_allocarray source code, 44
pbm_allocrow source code, 44
PBM format. *See also* Formats
  distinguishing between plain and raw formats, 61
  header
    components, 43
    reading, 60
    reading, low-level routines for, 62
    writing, 64
  initialization routines, 70
  initializing
    with pbm_init, 50
    with pm_init, 49
  overview and history, 35–42
  reading, 59
  single-bit, 36
    routines used for, 43
  startup code for applications, 48
  writing, 64
    an IMG file from, 148
PBM utilities, using together, 457
PBM_FORMAT_TYPE source code, 69
pbm_freearray, declaration, 44
pbm_freerow, declaration, 44
pbm_getbit source code, 59
pbm_getc source code, 62
pbm_getint source code, 63
pbm_getrawbyte source code, 63
pbm.h source code, 43
pbm_init source code, 50
pbm_readmagicnumber source code, 60
pbm_readpbm source code, 62
pbm_readpbminit source code, 60
pbm_readpbminitrest source code, 60
pbm_readpbmrow source code, 61
pbm_writepbm source code, 67
pbm_writepbminit source code, 64
pbm_writepbmrow source code, 66
pbm_writepbmrowplain source code, 66
pbm_writepbmrowraw source code, 65
pbmplus.h source code, 38
pbmtogem.c source code, 149
pbmtolj.c source code, 431
PCL (HP Printer Control Language), 431–454. *See also* Formats
  as extended file format, 11
  how to detect, 476
PCX format, 121–142. *See also* Formats
  encoding, 134, 138
  reading files in, 122–131
  writing files in, 131–142
pcx_planes_to_pixels source code, 128
pcx_unpack_pixels source code, 129
PCXEncode source code, 135
Pen plotters, HP-GL control language for, 407
PGM format, 36. *See also* Formats
  headers, 68
  reading, 71–74
  writing, 74–78
pgm_allocarray source code, 69
pgm_allocrow source code, 69
PGM_BIGGRAYS parameter, 68
pgm_freearray source code, 69
pgm_freerow source code, 69
pgm.h source code, 68
pgm_init source code, 71
pgm_readpgm source code, 74
pgm_readpgminit source code, 70, 71
pgm_readpgminitrest source code, 72
pgm_readpgmrow source code, 73
pgm_writepgm source code, 78
pgm_writepgminit source code, 75
pgm_writepgmrow source code, 77
pgm_writepgmrowplain source code, 76
pgm_writepgmrowraw source code, 77
Pixar RLE compression, 361. *See also* Compression techniques
pixarray datatype
  handling MS-DOS extended memory with, 24–30
  skeleton virtual memory pixel array routine source code, 27
Pixels
  color, scaling, 81
  nonsquare, warning the user if file contains, 159
  ordering, 7, 9
  lossy compression, 10
  representation
    grayscale, 68
    in PCX format, 138
    in PPM file format, 79
  term definition, 7
  virtual segmented pixel arrays in C++, 24 –30
PixMap array, representing pixels in PCX format with, 138
Pixval, representations in PPM file format, 79
pm_allocarray source code, 47
pm_allocrow source code, 46
pm_bitstomaxval source code, 54
pm_close source code, 55
pm_error source code, 52
pm_freearray source code, 47

pm_freerow source code, 46
pm_init source code, 49
pm_keymatch source code, 48
pm_maxvaltobits source code, 53
pm_message source code, 51
pm_openr source code, 54
pm_openw source code, 55
pm_perror source code, 51
pm_readbiglong source code, 56
pm_readbigshort source code, 56
pm_readlittlelong source code, 58
pm_readlittleshort source code, 57
pm_usage source code, 51
pm_writebiglong source code, 57
pm_writebigshort source code, 56
pm_writelittlelong source code, 58
pm_writelittleshort source code, 58
PMB_FORMAT_TYPE source code, 44
PNM format, 102–111. *See also* Formats
  header files, 102
  initializing, 104
  programs, 37
  reading, 104–107
  writing, 107–111
pnm_allocarray declaration, 103
pnm_allocrow declaration, 103
pnm_freearray declaration, 103
pnm_freerow declaration, 103
pnm.h source code, 103, 437
pnm_init source code, 104
pnm_readpnm source code, 107
pnm_readpnminit source code, 105
pnm_readpnmrow source code, 106
pnm_writepnm source code, 110
pnm_writepnminit source code, 108
pnm_writepnmrow source code, 109
Poskanzer
  aNy Map programs. *See* PNM programs
  formats. *See* PBM format; PGM format; PPM format
PostScript, 6, 11, 436–454
  reading, 451
  writing, 436
PPM format, 37. *See also* Formats
  converting
    BMP files to, 180
    HP-GL to, 470
    PCX file to, 122, 132
  initializing, 82
  libraries implementing, 79– 101
  reading, 83–87
  writing, 88–92
ppm_addtocolorhash source code, 94, 98
ppm_addtocolorhist source code, 93, 95
ppm_alloccolorhash source code, 94, 98
PPM_ASSIGN source code, 80
ppm_colorhashtocolorhist source code, 94, 99
ppm_colorhisttocolorhash source code, 94, 100
ppm_computecolorhash source code, 94, 97
ppm_computecolorhist source code, 93, 95
PPM_DEPTH source code, 82
PPM_EQUAL source code, 80
ppm_freecolorhash source code, 94, 101
ppm_freecolorhist source code, 101
PPM_GETB source code, 80
PPM_GETG source code, 80
PPM_GETR source code, 80
ppm.h source code, 79
ppm_init source code, 83, 123
ppm_lookupcolor source code, 94, 101
PPM_LUMIN macro source code, 82
PPM_MAXMAXVAL source code, 80
ppm_readppm source code, 87
ppm_readppminit source code, 83
ppm_readppminitrest source code, 82, 84
ppm_readppmrow source code, 85
ppm_writeppm source code, 92
ppm_writeppminit source code, 88
ppm_writeppmrow source code, 91
ppm_writeppmrowplain source code, 90
ppm_writeppmrowraw source code, 89
ppmcmap.h source code, 93
ppmd_circle source code, 463, 469
ppmd_line source code, 463, 465
ppmd_point_drawproc source code, 463, 464
ppmd_setlineclip source code, 464
ppmd_setlinetype source code, 464
ppmdraw.h source code, 463
ppmtobmp.c source code, 186
ppmtogif.c source code, 218
ppmtoilbm.c source code, 166–173
ppmtopcx.c source code, 131
ppmtotga.c source code, 204
Printers
  control languages for,
    PCL, 431–454
    PostScript, 6, 11, 436–454
  data files, 11
process_image source code, 24
prototypes.h source code, 275
psidtopgm.c source code, 452
put_big_short source code, 174
put_big_long source code, 174

put_byte source code, 174
put_chunk_head source code, 174
put_color_map source code, 400
put_fourchars source code, 174
put_map source code, 214
put_map_entry source code, 211
put_mono source code, 213
put_pixel source code, 213
put_pixel_rows source code, 400
put_rgb source code, 214
putbit source code, 152, 434
Putbyte source code, 141
putexlval source code, 447
putinit source code, 151, 434, 437
putitem source code, 152, 435, 447
PutPlane source code, 140
putrest source code, 152, 435, 448
putrow source code, 152
putus source code, 89
Putword source code, 141, 224

## Q

Quantization
converting full-color to lesser color versions, 458
major lossy step in JPEG encoding, 388

## R

random definition, 40
Rasterization, 12, 462–474
algorithms, 12
journal reference, 477
Raw files. *See also* Binary format; Files
PBM
distinguishing between plain PBM files and, 61
header designation, 43
reading a single byte, 63
writing, 65
PPM
reading, 85
writing, 89
read_JPEG_file source code, 402
read_pcx_image source code, 127
readBMP source code, 181
ReadColorMap source code, 244
ReadGIF source code, 241
readHeader source code, 423
readhpgl.cpp source code, 414
ReadImage source code, 252
reading source code, 29
ReadMacPaintFile source code, 116
readObject source code, 424
ReadOK source code, 239
ReadPlanes source code, 138
readtga source code, 199
readwmf.c source code, 423
ResetFieldBit source code, 268
RGB. *See* Color, full-color images
rindex definition, 40
RLE compression. *See* Compression techniques, RLE
rleflush source code, 450
rleputbuffer source code, 450
rleputitem source code, 448
rleputrest source code, 451
rleputxelval source code, 448
RowBytes source code, 155
Rows. *See* Memory management
Run-length encoding. *See also* Compression techniques
GEM IMG, 143–153
IFF/ILBM, 154–176
MacPaint, 115
PackBits, 17
PCX, 121–142

## S

Scan lines, 7
Scanners, effect on pixel ordering, 8
Screen coordinates, 5
sendbytes source code, 120
setFloatArray source code, 283
setLongArray source code, 282
setpixel source code, 22
setShortArray source code, 282
setString source code, 282
Silicon Graphics RLE compression, 361. *See also* Compression techniques
Spatial frequencies, 387
srandom definition, 40
Startup code, PBM applications, 48
Strips, 30

## T

Tag types, TIFF, 255
Targa format, 192–214. *See also* Formats
how to detect, 476
writing, 203–214

Text
character, reading, 62
labels, implementing in the HP-GL to PPM converter, 474
TGA format. *See* Targa file format
`tga.h` source code, 192
`tgatoppm.c` source code, 193
Thunderscan compression, 361. *See also* Compression techniques
`tif_aux.c` source code, 296
`tif_cleanup` function, 346
`tif_close` function, 346
`tif_compat.c` source code, 377
`tif_compress.c` source code, 346
`tif_decoderow` function, 345
`tif_decodestrip` function, 346
`tif_decodetile` function, 346
`tif_dir.c` source code, 281
`tif_dirinfo.c` source code, 277–280
`tif_dirread.c` source code, 298
`tif_dirwrite.c` source code, 320
`tif_encoderow` function, 346
`tif_encodestrip` function, 346
`tif_encodetile` function, 346
`tif_error.c` source code, 381
`tif_open.c` source code, 336
`tif_packbits.c` source code, 355
`tif_postencode` function, 345
`tif_predecode` function, 345
`tif_preencode` function, 345
`tif_seek` function, 346
`tif_swab.c` source code, 377
`tif_warning.c` source code, 382
TIFF (Tagged Image File Format), 255–384. *See also* Formats
advantages and disadvantages, 3
Baseline, 255
closing, 343–345
data
reading, 361–367
writing, 368–376
fields used for reading and writing image data, 351
file format, 255
I/O library headers
source code, private library, 265–267
source code, public library, 263–265
JPEG-in-TIF F subformat, 386
library, 256–298
headers, 257
LZW compression use by, 19
opening and closing, 335
reading
algorithm for, 256
application source code, 383
subsets and classes, 255
tag datatype definitions, 259–263
writing, algorithm for, 256
`tiff` structure source code, 269
`tiff.h` source code, 257
`TIFFAppendToStrip` source code, 376
`TIFFBitRevTable` source code, 379
`TIFFBufferSetup` source code, 374
`TIFFCheckRead` source code, 363
`TIFFClose` source code, 344
`TIFFClrFieldBit` source code, 268
`tiffcompat.h` source code, 272
`TIFFCurrentDirectory` source code, 343
`TIFFCurrentRow` source code, 343
`TIFFCurrentStrip` source code, 343
`TIFFCurrentTile` source code, 343
`tiffDataWidth` source code, 280
`TIFFDefaultDirectory` source code, 295
`TIFFError` source code, 382
`TIFFErrorHandler` source code, 382
`TIFFExtractData` source code, 268
`TIFFFdOpen` source code, 339
`TIFFFetchByteArray` source code, 313
`TIFFFetchData` source code, 311
`TIFFFetchExtraSamples` source code, 320
`TIFFFetchFloat` source code, 313
`TIFFFetchFloatArray` source code, 315
`TIFFFetchLongArray` source code, 314
`TIFFFetchNormalTag` source code, 316–318
`TIFFFetchPerSampleShorts` source code, 318
`TIFFFetchRational` source code, 312
`TIFFFetchRationalArray` source code, 315
`TIFFFetchShortArray` source code, 314
`TIFFFetchString` source code, 311
`TIFFFetchStripThing` source code, 318
`tiffFieldInfo` array
component descriptions, 276
source code, 277–280
`TIFFFieldInfo` source code, 268
`TIFFFieldSet` source code, 268
`TIFFFieldWithTag` source code, 281
`TIFFFileName` source code, 342
`TIFFFileno` source code, 342
`TIFFFillStrip` source code, 365
`TIFFFindFieldInfo` source code, 281
`TIFFFlush` source code, 344
`TIFFFlushData` source code, 345, 375
`TIFFFreeDirectory` source code, 296

TIFFGetField source code, 290–293, 293
_TIFFGetField source code, 294
TIFFGetFieldDefaulted source code, 298
TIFFGetFileSize source code, 377
TIFFGetMode source code, 342
TIFFGrowStrips source code, 374
TIFFInitDumpMode source code, 351
TIFFInitOrder source code, 337
TIFFInitPackBits source code, 355
TIFFInsertData source code, 268
tiffio.h source code, 263, 270
tiffioP.h source code, 265–267
TIFFIsTiled source code, 342
TIFFLinkDirectory source code, 334
TIFFNoBitRevTable source code, 380
TIFFNoDecode source code, 349
TIFFNoEncode source code, 348
TIFFNoRowDecode source code, 349
TIFFNoRowEncode source code, 348
TIFFNoStripDecode source code, 349
TIFFNoStripEncode source code, 349
TIFFNoTileDecode source code, 350
TIFFNoTileEncode source code, 349
TIFFOpen source code, 338
TIFFReadBufferSetup source code, 366
TIFFReadDirectory source code, 300–309
TIFFReadRawStrip source code, 367
TIFFReadScanline source code, 362
TIFFReverseBits source code, 381
TIFFScanlineSize source code, 341
TIFFSeek source code, 363
TIFFSetCompressionScheme source code, 350
TIFFSetField source code, 283, 289
TIFFSetFieldBit source code, 268
TIFFSetupShortLong source code, 329
TIFFSetupShortPair source code, 330
TIFFSetupStrips source code, 373
TIFFSetWarningHandler source code, 383
TIFFStartStrip source code, 367
TIFF SwabArrayOfLong source code, 378
TIFFSwabArrayOfShort source code, 378
TIFFSwabLong source code, 378
TIFFSwabShort source code, 377
TIFFVGetField source code, 293
TIFFVGetFieldDefaulted source code, 296–298
TIFFVSetField source code, 289
TIFFWarning source code, 383
TIFFWriteCheck source code, 371–373
TIFFWriteData source code, 334
TIFFWriteDirectory source code, 321–326
TIFFWriteFloatArray source code, 333
TIFFWriteLongArray source code, 332
TIFFWriteNormalTag source code, 327
TIFFWritePerSampleShorts source code, 330
TIFFWriteRational source code, 329
TIFFWriteRationalArray source code, 333
TIFFWriteScanline source code, 368–371
TIFFWriteShortArray source code, 332
TIFFWriteShortTable source code, 331
TIFFWriteString source code, 331
Tiles, 30
trace_message source code, 398
Truevision, Targa file format, 192–214

## U

Uncompressed formats
  Targa, 192–214
  Windows Bitmap (BMP), 177–191
Uncompressing, PCX images, 125
  with read_pcx_image, 127
unlockrow source code, 29
Unpacking. *See* Uncompressing

## V

Variable-length arrays, library routines supporting, 45
Vector format, 5
  advantages and disadvantages, 10
  conversion, 462–474
  memory management techniques, 30
  Postscript handling of, 11
version.h source code, 42
vfprintf emulation, 53
Video
  adapters, pixel ordering, 9
  frames, HAM mode use for, 155
Virtual memory
  allocating, 27
  deallocating, 28
  handling large files effectively in, 23
  segmented pixel arrays, in C++, 24–30
vpixel.ccp source code, 27

## W

Windows
  metafiles. *See* WMF (Windows metafiles)
  Windows Bitmap (BMP) format. *See* BMP file format

WMF (Windows metafiles) format, 418–427. *See also* Formats
  advantages and disadvantages, 418–427
  reading, in C, 420
  reading and writing, in C++, 419–427
`wmf.h` source code, 420
`write_JPEG_file` source code, 393
`writeBMP` source code, 188
`writeerr` declaration, 219
`writemacp` source code, 118
`WriteRationalPair` source code, 321
`writetga` source code, 210
`writing` source code, 29

## X

Xel type, 102
Xelval type, 102

## Z

Zsoft Corporation, PCX file format designed by, 121

The PBM, PPM, PGM, and PNM routines in Chapters 6–14, 16, 17, 22–25:

Copyright © 1988, 1989, 1991 by Jef Poskanzer.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided AS IS without express or implied warranty.

Parts of the MacPaint routines in Chapter 11:

Copyright © 1987 by Patrick J. Naughton. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

The PCX routines in Chapter 12:

Copyright © 1990 by Michael Davidson.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

This file is provided AS IS with no warranties of any kind. The author shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by this file or any part thereof. In no event will the author be liable for any lost revenue or profits or other special, indirect and consequential damages.

The GEM IMG reading program in Chapter 13:

Author: Diomidis D. Spinellis

Copyright © 1988 by Diomidis D. Spinellis.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

This file is provided AS IS with no warranties of any kind. The author shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by this file or any part thereof. In no event will the author be liable for any lost revenue or profits or other special, indirect and consequential damages.

The GEM IMG writing program in Chapter 13:

Author: David Beckemeyer (bdt!david)

Copyright © 1988 by David Beckemeyer (bdt!david) and Jef Poskanzer.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided AS IS without express or implied warranty.

GIF Format, used in Chapter 17:

The Graphics Interchange Format (c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

The GIF reading program in Chapter 17:

Copyright © 1990 by David Koblas. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above

copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided AS IS without express or implied warranty.

The TIFF library in Chapter 18:

Copyright © 1988, 1989, 1990, 1991, 1992 by Sam Leffler

Copyright © 1991, 1992 by Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that (i) the above copyright notices and this permission notice appear in all copies of the software and related documentation, and (ii) the names of Sam Leffler and Silicon Graphics may not be used in any advertising or publicity relating to the software without the specific, prior written permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The TIFF conversion applications in Chapter 18:

Derived by Jef Poskanzer from tif2ras.c, and ras2tif.c

Copyright © 1990 by Sun Microsystems, Inc.

Author: Patrick J. Naughton, naughton@wind.sun.com

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

This file is provided AS IS with no warranties of any kind. The author shall have no liability with respect to the infringement of copyrights, trade secrets or any patents by this file or any part thereof. In no event will the author be liable for any lost revenue or profits or other special, indirect and consequential damages.

The Laserjet programs in Chapter 22:

Copyright © 1988 by Jef Poskanzer and Michael Haberler.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided AS IS without express or implied warranty.

# Ordering the Companion Disk

If you bought *Programming for Graphic Files in C and C++* without the Companion Disk, you can order the disk by mail or phone. The disk contains all the source code listed in the book, as well as related code, including the Poskanzer Bitmap Utilities, the Leffler TIFF Library, and the Independent JPEG Group software. (For a detailed description of the contents of the Companion Disk, see page 478.)

To order any of the items listed below, fill out the form and mail it to the address indicated or telephone your order directly to John Wiley & Sons, Inc. at **1-800-879-4539**. Be sure to be ready with your credit card information and the ISBN listed for each item.

---

**DISK** (DOS, High Density, 3.5") Please send me _____ copy(ies) of the Companion Disk for use with the book. *Levine: Programming for Graphics Files in C and C++* at $20.00 each. ISBN 0-471-59857-7.

**BOOK** Please send me _____ copy(ies) of the book *Levine: Programming for Graphics Files in C and C++* at $29.95 each. ISBN 0-471-59854-2.

**BOOK/DISK SET** Please send me _____ copy(ies) of the book/disk set, *Levine: Programming for Graphics Files in C and C++* with the Companion Disk at $49.95 each. ISBN 0-471-59856-9.

☐ Payment Enclosed ☐ Visa ☐ MasterCard ☐ American Express

Card Number _____ Expiration Date _____

Signature _____

Name _____

Company Name _____

Address _____

City/State _____ Zip Code _____

Mail your order to:

JOHN WILEY & SONS, INC.
Attn: Continuation Department
P.O. Box 6792
Somerset, NJ 08875-9976

DATE DUE

| | | | |
|---|---|---|---|
| NOV 2 4 '97 | | | |
| FEB 2 0 '98 | | | |
| MAR 1 5 '98 | | | |
| APR 1 5 '98 | | | |
| OCT 1 3 '99 | | | |
| DE 1 0 '03 | | | |
| | | | |
| | | | |
|

T385 .L47 1994
Levine, John R
Programming for graphics
files in C and C++

WESTERN CAROLINA U.-HUNTER LIB
3 0470 0574437 1

C PROGRAMMING/GRAPHICS $29.95

# PROGRAMMING FOR GRAPHICS FILES IN C AND C++

Even the best programmers can find the details of handling graphics files to be a tricky and often frustrating experience. You have to resolve memory management, format conversion, file compression, and a host of other complex issues before you can even think about reading, writing, and processing graphics files.

This technical source book gives C and C++ programmers the tools they need to incorporate efficient graphics-handling routines in their code, including:

- Compression techniques for very large graphics files, with detailed analyses of the most popular and innovative schemes, including JPEG
- Memory-management strategies for segmented (DOS) and virtual (Macintosh, UNIX, NT, and OS/2) environments
- Algorithms and code for format conversions, as well as a guide to reconciling file formats with printer formats
- A special framework for bitmap files
- Platform-independent C and C++ source code

**About the author**

**JOHN LEVINE** is an independent software consultant, writer, and the author of four previous computer titles. He has a Ph.D. in computer science from Yale University.

*Art Direction: José Almaguer*
*Cover Art & Design: Mona Mark*

**John Wiley & Sons, Inc.**
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0-471-59854-2